目錄

Redis开发运维实践指南

本手册是我在一家中国大陆的中型商业银行做大数据系统工程师中进行的总结归纳，包含开发和运维的各方面的使用、应用场景和最佳实践，以及各个高可用架构的搭建和测试。

下载和线上阅读最新版请访问：https://www.gitbook.com/book/gnuhpc/redis-all-about/details

我会定期持续更新此指南，以供各位了解认识redis。

其中会有引用的各种文献和图片，我会尽可能的标注引用，如果您发现您的文字或者图片未标注引用，请fork本手册的github并pull request： https://github.com/gnuhpc/All-About-Redis

有不正确的地方，请通过gnuhpc@weixin或者huangpengcheng#cmbc.com.cn (:s/#/@)联系我。

如果你觉得这些内容有帮助，欢迎Star我，也欢迎打赏 ^_^ ，我会不定期公布支持我的好朋友名单。

— Pengcheng Huang

新书发布：

我和Airbnb美国的工程师王左非一起写的新书Redis 4.x Cookbook已经由Packt出版，目前在Amazon.com上销售，请大家多多支持！ https://www.amazon.com/Redis-4-X-Cookbook-Pengcheng-Huang/dp/1783988169/ref=sr_1_2?ie=UTF8&qid=1519899713&sr=8-2&keywords=redis&dpID=51x10vJb%252B6L&preST=_SX218_BO1,204,203,200_QL40_&dpSrc=srch

亚马逊中国Kindle版地址： https://www.amazon.cn/dp/B07B6SDXQH/ref=sr_1_2?ie=UTF8&qid=1522737150&sr=8-2&keywords=redis+cookbook

中文版翻译已经完成，已于博文视点出版, 欢迎大家购买： https://item.jd.com/12364212.html

在豆瓣上欢迎大家点评，并指出错误，谢谢！ https://book.douban.com/subject/30227261/

技术交流：

谢谢大家的赞助，我建立了一个Redis技术交流微信群（主群已满，有目前codis、pika、tidb等知名开源软件的作者），专门交流Redis开发和运维相关技术的，由于已经超100人，想加入的请先加我的微信：gnuhpc。谢谢！

技术交流记录：

为了上述Redis技术交流群，我还创建了一个知识星球，您可以使用微信扫描下列二维码进



入，免费

长期招聘：

我现在所在团队负责我行全行大数据基础架构平台和产品的建设和维护工作，待遇大于等于一线互联网，技术实力银行业大数据基础架构前三名，较之同规模的互联网团队也有技术特点。

坐标：北京市顺义区（公司有市内多趟班车来往，开车反方向无压力，团队崇尚任务驱动型，基本不加班）。

欢迎期待工作生活平衡又希望技术上有所精进的同学联系，有意者请发简历至gnuhpc@foxmail.com 或加微信gnuhpc，具体要求如下：

总行部门拟聘岗位职责及任职资格说明

岗位一	数据库管理岗（大数据）		
招聘渠道	√社会招聘 √行内招聘 √校园招聘 √猎头	招聘人数	3
岗位职责（简短有序）	1、负责大数据平台架构的研究设计，参与设计、研发大数据处理平台。平台包括海量数据存储和计算、实时流式计算和高效任务调度等功能。 2、负责处理大数据相关产品的故障和缺陷。 3、负责大数据平台的运维和性能监控与调优。 4、基于开源大数据产品增强符合行内需要有关功能，包括核心功能和一些管理功能； 5、负责大数据平台的生产维护和应急处理。 6、负责大数据技术生产的各种标准和规则的制定及实施。 7、负责研究与推广大数据相关技术。		
任职资格	1、计算机等相关专业，本科及以上学历，两年以上相关技术领域工作经验； 2、可以熟练使用Java/Scala/python编程语言的一种，熟练Java分布式开发者优先。 3、具有大数据领域相关研究背景，对Hadoop、Spark、分布式系统、实时流计算Spark Streaming/Storm等技术种的一个或多个有较深的理解，熟悉Hadoop/Spark生态系统各功能组件、熟悉源码者优先。对熟练掌握主流实时流计算数据采集、分析处理架构的优先。 4、具备快速学习掌握新知识的能力，优秀的分析、解决问题能力；具备良好的抽象归纳能力和创新能力； 5、有强烈的责任感，主动性强，团队协作和沟通能力强。 6、身体健康，较强的自我计划和实施执行的能力，激励、沟通、协调能力，具备良好的职业操守和强烈的责任心。 7、熟悉开源社区的使用和参与，并在开源社区有贡献者优先。		

赞助名单（由于微信面对面收钱无法显示，停止更新）：

1. Tina 10元 2016-01-13
2. FloydZhang 10元 2016-01-15
3. Sammy-Ip 16.8元 2016-01-21
4. Yunwei 10.1元 2016-01-25
5. csyangchsh 15元 2016-01-29
6. 黄海斌 10元 2016-04-04
7. Ghost 16.88元 2016-04-13
8. zbdba 18.8元 2016-04-19
9. 慎独 10元 2016-04-19
10. Truman 18.8元 2016-04-20
11. 蓝金伟 8.8元 2016-04-26
12. Mengjie 18.8元 2016-04-26
13. Jason Young 66元 2016-04-26

....

微信二维码：

微信扫一扫转账

向鹏程转账

支付宝二维码：

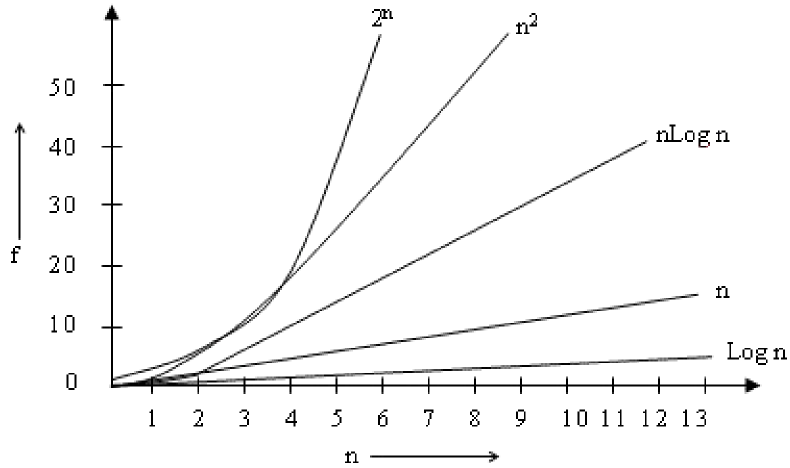

黄鹏程
北京 顺义
用支付宝扫二维码，加我好友

1. 简述

Redis为一个运行在内存中的数据结构服务器（data structures server）。Redis使用的是单进程（除持久化时），所以在配置时，一个实例只会用到一个CPU。本手册分为三部分，第一部分从开发角度介绍了redis开发中使用API、场景和生产设计规范最佳实践，第二部分从运维角度介绍了如何运维redis，其间的常见操作和最佳实践等,第三部分是从高可用和集群方面介绍了redis相关的集群架构、搭建和测试。

2. 数据操作

熟悉每个数据操作前一定要明白每个操作都是代价，以时间复杂度和对应查询集或者结果集大小为衡量。时间复杂度收敛状况如下：



2.1. key操作

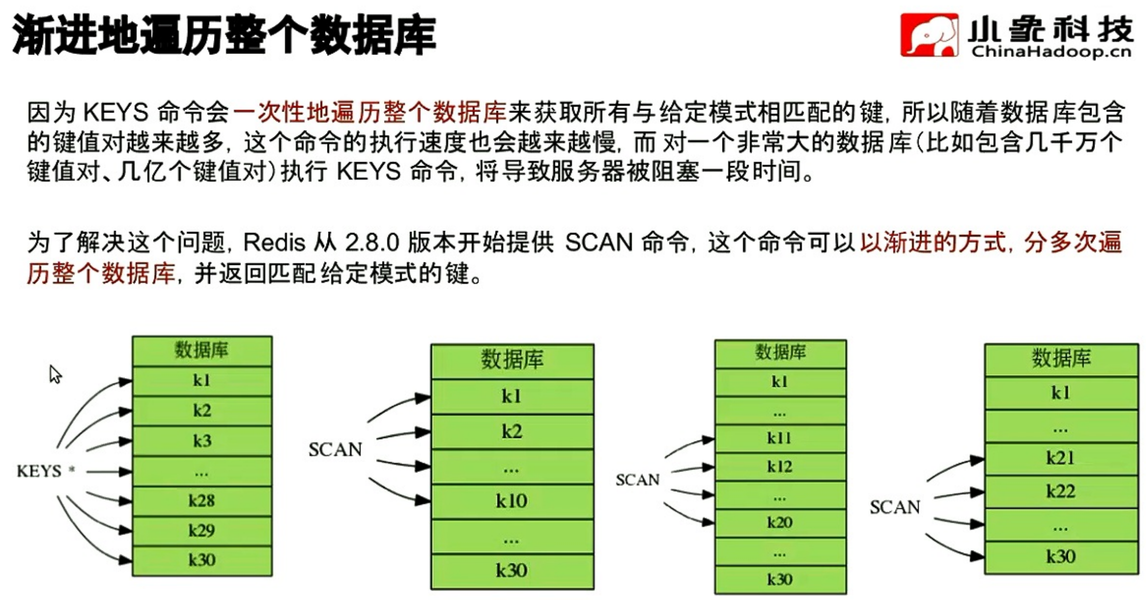
redis 的key操作是涉及范围最广的操作 。

2.1.1 列出key

```
keys  *user*
keys  *
```

有3个通配符 *, ? ,[]

- *: 通配任意多个字符
- ?: 通配单个字符
- []: 通配括号内的某1个字符

> 注：生产已经禁止。更安全的做法是采用scan，原理和操作如下：
>
> 
>
>
> 针对Keys的改进，支持分页查询Key。在迭代过程中，Keys有增删时不会要锁定写操作，数据集完整度不做任何保证，同一条key可能会被返回多次.
>
> | | KEYS | SCAN |
> |---|---|---|
> | **处理方式** | 一次性遍历整个数据库并返回所有结果。 | 每次遍历数据库中的一部分键，并返回一部分结果。 |
> | **是否会阻塞服务器？** | 可能会 | 不会 |
> | **是否会出现重复值？** | 不会 | 可能会 |
> | **复杂度** | O(N)，N 为数据库包含的键值对数量。 | 每次执行的复杂度为 O(1)，遍历整个数据库的总复杂度为 O(N)，N 为数据库包含的键值对数量。 |
>
> 对于其他危险的命令，新版本也进行了替代：

> # 其他渐进遍历命令
>
> SSCAN key cursor [MATCH pattern] [COUNT count]
>
> 代替可能会阻塞服务器的 SMEMBERS 命令，遍历集合包含的各个元素。
>
> HSCAN key cursor [MATCH pattern] [COUNT count]
>
> 代替可能会阻塞服务器的 HGETALL 命令，遍历散列包含的各个键值对。
>
> ZSCAN key cursor [MATCH pattern] [COUNT count]
>
> 代替可能会阻塞服务器的 ZRANGE 命令，遍历有序集合包含的各个元素。
>
> 这些命令的 MATCH 选项和 COUNT 选项的意义和 SCAN 命令的一样。

redis-cli下的扫描:

```
redis-cli --scan --pattern 'chenqun_*'
```

这是用scan命令扫描redis中的key，--pattern选项指定扫描的key的pattern。相比keys *pattern* 模式,不会长时间阻塞redis而导致其他客户端的命令请求一直处于阻塞状态。

> 本页中采用了小象学院的两张片子，版权归 http://www.chinahadoop.cn/ 所有

2.1.2 测试指定key是否存在

```
exists key
```

返回1表示存在，0不存在

2.1.3 删除给定key

```
del key1 key2 ....keyN
```

返回1表示存在，0不存在

2.1.4 返回给定key的value类型

```
type key
```

返回 none 表示不存在key。string字符类型，list 链表类型 set 无序集合类型...

2.1.5 返回从当前数据库中随机选择的一个key

```
randomkey
```

如果当前数据库是空的，返回空串

2.1.6 原子的重命名一个key

```
rename oldkey newkey
```

如果newkey存在，将会被覆盖，返回1表示成功，0失败。可能是oldkey不存在或者和newkey相同

```
renamenx oldkey newkey
```

同上，但是如果newkey存在返回失败

2.1.7 Key的超时设置处理

```
expire key seconds
```

单位是秒。返回1成功，0表示key已经设置过过期时间或者不存在。 如果想消除超时则使用persist key。如果希望采用绝对超时，则使用expireat命令。

```
ttl key
```

返回设置过过期时间的key的剩余过期秒数 -1表示没有设置过过期时间，对于不存在的key,返回-2。

```
pexpire key 毫秒数
```

设置生命周期。

```
pttl  key
```

以毫秒返回生命周期。

> 注意：
>
> 当client主动访问key会先对key进行超时判断，过时的key会立刻删除。
>
> 如果clien永远都不再get那条key呢？ 它会在Master的后台，每秒10次的执行如下操作：随机选取100个key校验是否过期，如果有25个以上的key过期了，立刻额外随机选取下100个key(不计算在10次之内)。可见，如果过期的key不多，它最多每秒回收200条左右，如果有超过25%的key过期了，它就会做得更多，但只要key不被主动get，它占用的内存什么时候最终被清理掉只有天知道。
>
> 在主从复制环境中，由于上述原因存在已经过期但是没有删除的key，在主snapshot时并不包含这些key，因此在slave环境中我们往往看到dbsize较master是更小的。

最大字符串为512M，但是大字符串非常不建议。

2.2.1 设置key对应的值为string类型的value

```
set key value [ex 秒数] / [px 毫秒数]  [nx] /[xx]
```

返回1表示成功，0失败

> 注: 如果ex,px同时写,以后面的有效期为准

```
setnx key value
```

仅当key不存在时才Set，如果key已经存在，返回0 。nx 是not exist的意思。

> 应用场景：用来选举Master或做分布式锁：所有Client不断尝试使用SetNx master myName抢注Master，成功的那位不断使用Expire刷新它的过期时间。如果Master倒掉了key就会失效，剩下的节点又会发生新一轮抢夺。

```
mset key1 value1 ... keyN valueN
```

一次设置多个key的值，成功返回1表示所有的值都设置了，失败返回0表示没有任何值被设置

```
msetnx key1 value1 ... keyN valueN
```

同上，但是不会覆盖已经存在的key

SET 命令还支持可选的 NX 选项和 XX 选项，例如：SET nx-str "this will fail" XX

- 如果给定了 NX 选项，那么命令仅在键 key 不存在的情况下，才进行设置操作；如果键 key 已经存在，那么 SET ... NX 命令不做动作（不会覆盖旧值）。
- 如果给定了 XX 选项，那么命令仅在键 key 已经存在的情况下，才进行设置操作；如果键 key 不存在，那么 SET ... XX 命令不做动作（一定会覆盖旧值）。在给定 NX 选项和 XX 选项的情况下，SET 命令在设置成功时返回 OK ，设置失败时返回 nil 。

2.2.2 获取key对应的string值

```
get key
```

如果key不存在返回nil

```
getset key value
```

原子的设置key的值，并返回key的旧值。如果key不存在返回nil。应用场景：设置新值，返回旧值，配合setnx可实现分布式锁。

分布式锁的思路：注意该思路要保证多台Client服务器的NTP一致。

1. C3发送SETNX lock.foo 想要获得锁，由于C0还持有锁，所以Redis返回给C3一个0
2. C3发送GET lock.foo 以检查锁是否超时了，如果没超时，则等待或重试。
3. 反之，如果已超时，C3通过下面的操作来尝试获得锁：
4. GETSET lock.foo
5. 通过GETSET，C3拿到的时间戳如果仍然是超时的，那就说明，C3如愿以偿拿到锁了。
6. 如果在C3之前，有个叫C4的客户端比C3快一步执行了上面的操作，那么C3拿到的时间戳是个未超时的值，这时，C3没有如期获得锁，需要再次等待或重试。留意一下，尽管C3没拿到锁，但它改写了C4设置的锁的超时值，不过这一点非常微小的误差带来的影响可以忽略不计。

伪代码为：

```
# get lock
lock = 0
while lock != 1:
    timestamp = current Unix time + lock timeout + 1
    lock = SETNX lock.foo timestamp
    if lock == 1 or (now() > (GET lock.foo) and now() > (GETSET lock.foo timestamp)):
        break;
    else:
        sleep(10ms)

# do your job
do_job()

# release
if now() < GET lock.foo:
    DEL lock.foo
```

以上是一个单Server 的分布式锁思路，官网上还介绍了另一个单机使用超时方式进行的思路，和这个基本一致，并且在同一个文档中介绍了一个名为redlock的多Server容错型分布式锁的算法，同时列出了多语言的实现。这个算法的优势在于几个服务器可以有少量的时间差，不要求严格时间一致。

也可以设计一个按小时计算的计数器，可以用GetSet获取计数并重置为0。

```
mget key1 key2 ... keyN
```

一次获取多个key的值，如果对应key不存在，则对应返回nil

```
incr key
```

对key的值做加加操作,并返回新的值。注意incr一个不是int的value会返回错误，incr一个不存在的key，则设置key为1。范围为64有符号，-9223372036854775808~9223372036854775807。

```
decr key
```

同上，但是做的是减减操作，decr一个不存在key，则设置key为-1

```
incrby key integer
```

同incr，加指定值 ，key不存在时候会设置key，并认为原来的value是 0

```
decrby key integer
```

同decr，减指定值。decrby完全是为了可读性，我们完全可以通过incrby一个负值来实现同样效果，反之一样。

```
incrbyfloat key floatnumber
```

针对浮点数

浮点数的自增和自减

INCRBYFLOAT key increment

为字符串键 key 储存的值加上浮点数增量 increment，命令返回操作执行之后，键 key 的值。

没有相应的 DECRBYFLOAT，但可以通过给定负值来达到 DECRBYFLOAT 的效果。

复杂度为 O(1)。

```
redis> SET num 10
OK
redis> INCRBYFLOAT num 3.14
"13.14"
redis> INCRBYFLOAT num -2.04   # 通过传递负值来达到做减法的效果
"11.1"
```

哪些可以被操作呢？

值	能否执行数字值命令？	原因
10086	可以	值可以被解释为整数
3.14	可以	值可以被解释为浮点数
+123	可以	值可以被解释为整数
123456789123456789123456789	不可以	值太大，没办法使用 64 位整数来储存
2.0e7	不可以	Redis 不解释以科学记数法表示的浮点数
123ABC	不可以	值包含文字
ABC	不可以	值为文字

这个操作的应用场景：计数器

2.2.4 追加字符串

```
append key value
```

返回新字符串值的长度。

2.2.5 截取字符串

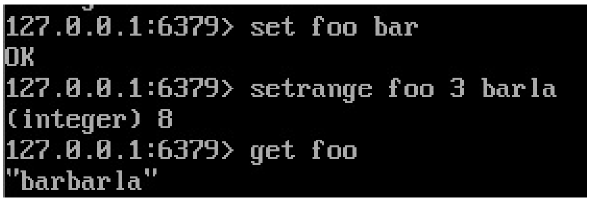
```
substr key start end
```

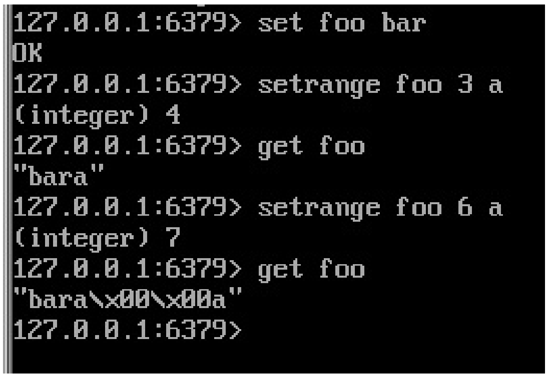
返回截取过的key的字符串值,注意并不修改key的值。下标是从0开始的

2.2.6 改写字符串

```
SETRANGE key offset value
```

用value 参数覆写(overwrite)给定key 所储存的字符串值，从偏移量offset 开始。 不存在的key 当作空白字符串处理。可以用作append：



> 注意: 如果偏移量>字符长度, 该字符自动补0x00，注意它不会报错
>
> 

2.2.7 返回子字符串

GETRANGE key start end

返回key 中字符串值的子字符串，字符串的截取范围由start 和end 两个偏移量决定(包括start 和end 在内)。可以使用负值，字符串右面下标是从-1开始的。

注意返回值处理：
1: start>=length, 则返回空字符串 2: stop>=length,则截取至字符结尾 3: 如果start 所处位置在stop右边, 返回空字符串

```
$ redis-cli --raw    # 在 redis-cli 中使用中文时，必须打开 --raw 选项，才能正常显示中文

redis> SET msg "世界你好"    # 设置四个中文字符
OK

redis> GET msg    # 储存中文没有问题
世界你好

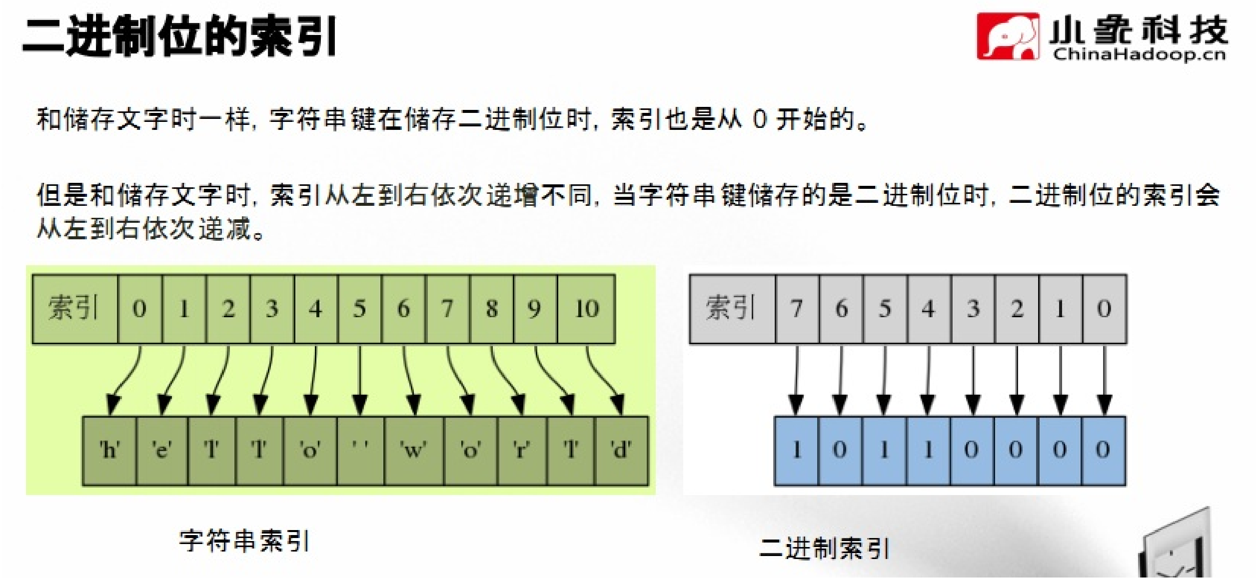
redis> STRLEN msg    # 这里 STRLEN 显示了“世界你好”的字节长度为 12 字节
12                     # 但我们真正想知道的是 msg 键里面包含多少个字符
```

2.2.9 取指定key的value值的长度

```
strlen
```

2.2.10 位操作

注意：位操作中的位置是反过来的，**offset**过大,则会在中间填充**0**，比如 **SETBIT bit 0 1**，此时**bit**为**10000000**，此时再进行**SETBIT bit 7 1**，此时**bit**为**10000001**。**offset**最大**2^32-1**。

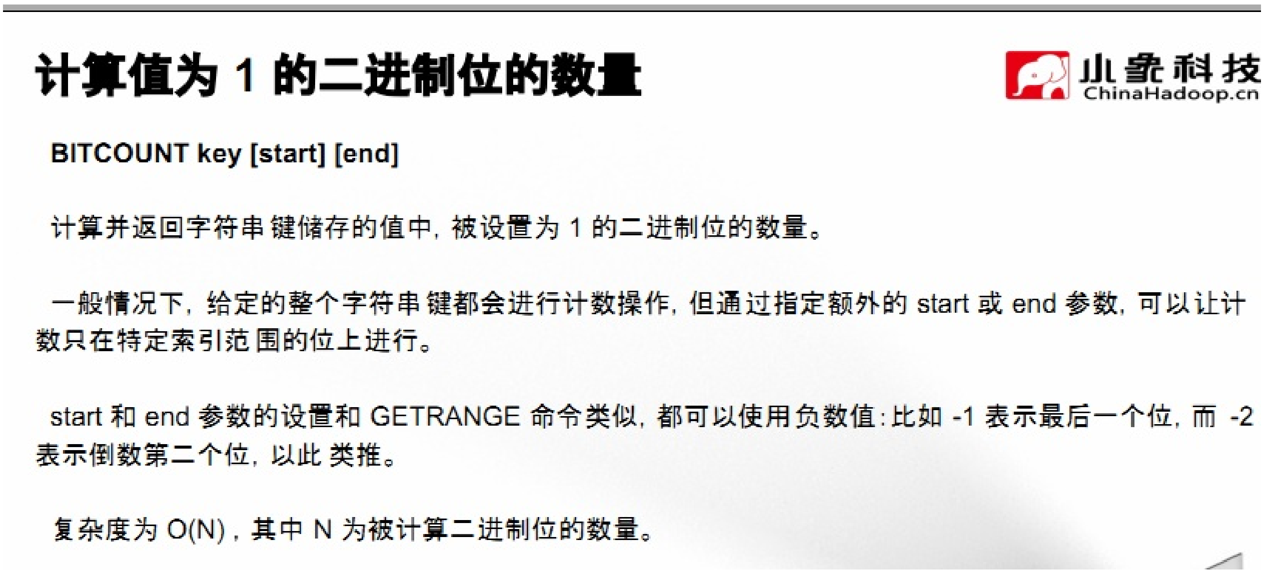



```
GETBIT key offset / SETBIT key offset value
```

设置某个索引的位为0/1

```
bitcount
```

对位进行统计



```
bitop
```

对1个或多个key对应的值进行AND/OR/XOR/NOT操作

注意:

> 1.bitop操作避免阻塞应尽量移到slave上操作. 2.对于NOT操作, key不能多个

2.3.1 添加元素

```
lpush key string
```

在key对应list的头部添加字符串元素，返回1表示成功，0表示key存在且不是list类型。注意：江湖规矩一般从左端Push，右端Pop，即LPush/RPop。

```
lpushx
```

也是将一个或者多个value插入到key列表的表头，但是如果key不存在，那么就什么都不在，返回一个false【rpushx也是同样】

```
rpush key string
```

同上，在尾部添加

```
linsert
```

在key对应list的特定位置之前或之后添加字符串元素，例如

```
redis 127.0.0.1:6379linsert mylist3 before "world" "there"
```

2.3.2 查看列表长度

```
llen key
```

返回key对应list的长度，key不存在返回0,如果key对应类型不是list返回错误

2.3.3 查看元素

```
lindex
```

返回名称为key的list中index位置的元素，例如：

```
redis 127.0.0.1:6379> lindex mylist5 0
```

2.3.4 查看一段列表

```
lrange key start end
```

返回指定区间内的元素，下标从0开始，负值表示从后面计算，-1表示倒数第一个元素 ，key不存在返回空列表。特殊的，

```
lrange key 0 -1
```

返回所有数据。

2.3.5 截取list

```
ltrim key start end
```

保留指定区间内元素，成功返回1，key不存在返回错误。O(N)操作。

> 注意：N是被移除的元素的个数，不是列表长度。

2.3.6 删除元素

```
lrem key count value
```

从key对应list中删除count个和value相同的元素。count为0时候删除全部，count为正，则删除匹配count个元素，如果为负数，则是从右侧扫描删除匹配count个元素。复杂度是O(N)，N是List长度，因为List的值不唯一，所以要遍历全部元素，而Set只要O(log(N))。

```
lpop key
```

从list的头部删除元素，并返回删除元素。如果key对应list不存在或者是空返回nil，如果key对应值不是list返回错误。

```
rpop
```

同上，但是从尾部删除。

2.3.7 设置list中指定下标的元素值

```
lset key index value
```

成功返回1，key或者下标不存在返回错误

2.3.8 阻塞队列

```
blpop key1...keyN timeout
```

从左到右扫描返回对第一个非空list进行lpop操作并返回，比如blpop list1 list2 list3 0 ,如果list不存在list2,list3都是非空则对list2做lpop并返回从list2中删除的元素。如果所有的list都是空或不存在，则会阻塞timeout秒，timeout为0表示一直阻塞。当阻塞时，如果有client对key1...keyN中的任意key进行push操作，则第一在这个key上被阻塞的client会立即返回（返回键和值）。如果超时发生，则返回nil。有点像unix的select或者poll。

```
brpop
```

同blpop，一个是从头部删除一个是从尾部删除。

> 注意：不要采用其作为ajax的服务端推送，因为连接有限，遇到问题连接直接打满。

BLPOP/BRPOP 的先到先服务原则 如果有多个客户端同时因为某个列表而被阻塞，那么当有新值被推入到这个列表时，服务器会按照先到先服务（first in first service）原则，优先向最早被阻塞的客户端返回新值。举个例子，假设列表 lst 为空，那么当客户端 X 执行命令 BLPOP lst timeout 时，客户端 X 将被阻塞。在此之后，客户端 Y 也执行命令 BLPOP lst timeout ，也因此被阻塞。如果这时，客户端 Z 执行命令 RPUSH lst "hello" ，将值 "hello" 推入列表 lst ，那么这个 "hello" 将被返回给客户端 X ，而不是客户端 Y ，因为客户端 X 的被阻塞时间要早于客户端 Y 的被阻塞时间。

rpoplpush/brpoplpush：rpoplpush srckey destkey 从srckey对应list的尾部移除元素并添加到destkey对应list的头部,最后返回被移除的元素值，整个操作是原子的.如果srckey是空或者不存在返回nil，注意这是唯一一个操作两个列表的操作，用于两个队列交换消息。

应用场景：task + bak 双链表完成工作任务转交的安全队列，保证原子性。 业务逻辑: 1: Rpoplpush task bak 2: 接收返回值,并做业务处理 3: 完成时用LREM消掉。如不成功或者如果集群管理(如zookeeper)发现worker已经挂掉,下次从bak表里取任务

另一个应用场景是循环链表： 127.0.0.1:6379> lrange list 0 -1 1) "c" 2) "b" 3) "a" 127.0.0.1:6379> rpoplpush list list "a" 127.0.0.1:6379> lrange list 0 -1 1) "a" 2) "c" 3) "b"

2.4.1 添加元素

```
sadd key member
```

成功返回1,如果元素以及在集合中返回0,key对应的set不存在返回错误

2.4.2 移除元素

```
srem key member
```

成功返回1，如果member在集合中不存在或者key不存在返回0，如果key对应的不是set类型的值返回错误

2.4.3 删除并返回元素

```
spop key
```

如果set是空或者key不存在返回nil

2.4.4 随机返回一个元素

```
srandmember key
```

同spop，随机取set中的一个元素，但是不删除元素

2.4.5 集合间移动元素

```
smove srckey dstkey member
```

从srckey对应set中移除member并添加到dstkey对应set中，整个操作是原子的。成功返回1,如果member在srckey中不存在返回0，如果key不是set类型返回错误

2.4.6 查看集合大小

```
scard key
```

如果set是空或者key不存在返回0

2.4.7 判断member是否在set中

```
sismember key member
```

存在返回1，0表示不存在或者key不存在

2.4.8 集合交集

```
sinter key1 key2...keyN
```

返回所有给定key的交集

```
sinterstore dstkey key1...keyN
```

同sinter，但是会同时将交集存到dstkey下

2.4.9 集合并集

```
sunion key1 key2...keyN
```

返回所有给定key的并集

```
sunionstore dstkey key1...keyN
```

同sunion，并同时保存并集到dstkey下

2.4.10 集合差集

```
sdiff key1 key2...keyN
```

返回所有给定key的差集

```
sdiffstore dstkey key1...keyN
```

同sdiff，并同时保存差集到dstkey下

2.4.11 获取所有元素

```
smembers key
```

返回key对应set的所有元素，结果是无序的，集合元素很多时会阻塞，生产上禁用！

Sorted Set的实现是hash table(element->score, 用于实现ZScore及判断element是否在集合内)，和skip list(score->element,按score排序)的混合体。 skip list有点像平衡二叉树那样，不同范围的score被分成一层一层，每层是一个按score排序的链表。

ZAdd/ZRem是O(log(N))，ZRangeByScore/ZRemRangeByScore是O(log(N)+M)，N是Set大小，M是结果/操作元素的个数。可见，原本可能很大的N被很关键的Log了一下，1000万大小的Set，复杂度也只是几十不到。当然，如果一次命中很多元素M很大那谁也没办法了。

2.5.1 添加元素

```
zadd key score member
```

添加元素到集合，元素在集合中存在则更新对应score。

2.5.2 删除元素

```
zrem key member
```

1表示成功，如果元素不存在返回0

```
zremrangebyrank key min max
```

删除集合中排名在给定区间的元素

```
zremrangebyscore key min max
```

删除集合中score在给定区间的元素

2.5.3 增加score

```
zincrby key incr member
```

增加对应member的score值，然后移动元素并保持skip list保持有序。返回更新后的score值，可以为负数递减

2.5.4 获取排名

```
zrank key member
```

返回指定元素在集合中的排名（下标，注意不是分数）,集合中元素是按score从小到大排序的

```
zrevrank key member
```

同上,但是集合中元素是按score从大到小排序

2.5.5 获取排行榜

```
zrange key start end
```

类似lrange操作从集合中去指定区间的元素。返回的是有序结果

---

zrevrange key start end 同上，返回结果是按score逆序的,如果需要得分则加上withscores

> 注：index从start到end的所有元素

2.5.6 返回给定分数区间的元素

```
zrangebyscore key min max
```

可以指定inf为无穷

2.5.7 返回集合中score在给定区间的数量

```
zcount key min max
```

2.5.8 返回集合中元素个数

```
zcard key
```

2.5.9 返回给定元素对应的score

```
zscore key element
```

2.5.10 评分的聚合

```
ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight] [AGGREGATE SUM|MIN|MAX]
```

例如：

```
127.0.0.1:6379> zrangebyscore votes -inf inf withscores
1) "sina"
2) "1"
3) "google"
4) "5"
5) "baidu"
6) "10"
127.0.0.1:6379> zrangebyscore visits -inf inf withscores
1) "baidu"
2) "1"
3) "google"
4) "5"
5) "sina"
6) "10"
127.0.0.1:6379> zunionstore award 2 visits votes weights 1 2 aggregate sum
(integer) 3
127.0.0.1:6379> zrangebyscore award -inf inf withscores
1) "sina"
2) "12"
3) "google"
4) "15"
5) "baidu"
6) "21"
```

一个小技巧是如果需要对评分进行倍加，则使用如下的方法：

```
127.0.0.1:6379>zrangebyscore visits -inf inf withscores
1) "baidu"
2) "1"
3) "google"
4) "5"
5) "sina"
6) "10"
127.0.0.1:6379>zunionstore visits 1 visits weights 2
(integer) 3
127.0.0.1:6379>zrangebyscore visits -inf inf withscores
1) "baidu"
2) "2"
3) "google"
4) "10"
5) "sina"
6) "20"
```

底层实现是hash table，一般操作复杂度是O(1)，要同时操作多个field时就是O(N)，N是field的数量。应用场景：土法建索引。比如User对象，除了id有时还要按name来查询。

可以有如下的数据记录: (String) user:101 -> {"id":101,"name":"calvin"...} (String) user:102 -> {"id":102,"name":"kevin"...} (Hash) user:name:index-> "calvin"->101, "kevin" -> 102

2.6.1 设置hash值

```
hset key field value
```

设置hash field为指定值，如果key不存在，则先创建。

```
hsetnx
```

设置hash field为指定值，如果 key 不存在，则先创建。如果 field已经存在，返回0，nx是not exist的意思。

2.6.2 获取hash值

```
hget key field
```

获取指定的hash field

```
hmget key filed1....fieldN
```

获取全部指定的hash filed

```
hmset key filed1 value1 ... filedN valueN
```

同时设置hash的多个field

2.6.3 递增某一个域的值

```
hincrby key field integer
```

将指定的hash filed 加上给定值

2.6.4 判断某一个域是否存在

```
hexists key field
```

测试指定field是否存在

2.6.5 删除域

```
hdel key field
```

删除指定的hash field

2.6.6 获取域的数量

```
hlen key
```

返回指定hash的field数量

2.6.7 获取所有的域名

```
hkeys key
```

返回hash的所有field

2.6.8 获取所有域的值

```
hvals key
```

返回hash的所有value

2.6.9 获取所有域名和值

```
hgetall
```

返回hash的所有filed和value

2.7 HyperLogLog操作

HyperLogLog主要解决大数据应用中的非精确计数（可能多也可能少，但是会在一个合理的范围）操作，它可以接受多个元素作为输入，并给出输入元素的基数估算值，基数指的是集合中不同元素的数量。比如 {'apple', 'banana', 'cherry', 'banana', 'apple'} 的基数就是 3 。HyperLogLog 的优点是，即使输入元素的数量或者体积非常非常大，计算基数所需的空间总是固定的、并且是很小的。在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。

关于这个数据类型的误差：在一个大小为12k的key所存储的hyperloglog集合基数计算的误差是%0.81.

参考文献：http://highscalability.com/blog/2012/4/5/big-data-counting-how-to-count-a-billion-distinct-objects-us.html

2.7.1 将元素添加至 HyperLogLog

```
PFADD key element [element ...]
```

这个命令可能会对 HyperLogLog 进行修改，以便反映新的基数估算值，如果 HyperLogLog 的基数估算值在命令执行之后出现了变化， 那么命令返回 1 ， 否则返回 0 。 命令的复杂度为 O(N) ，N 为被添加元素的数量。

2.7.2 返回给定 HyperLogLog 的基数估算值

```
PFCOUNT key [key ...]
```

当只给定一个 HyperLogLog 时，命令返回给定 HyperLogLog 的基数估算值。当给定多个 HyperLogLog 时，命令会先对给定的 HyperLogLog 进行并集计算，得出一个合并后的 HyperLogLog ，然后返回这个合并 HyperLogLog 的基数估算值作为命令的结果（合并得出的 HyperLogLog 不会被储存，使用之后就会被删掉）。 当命令作用于单个 HyperLogLog 时，复杂度为 O(1) ， 并且具有非常低的平均常数时间。 当命令作用于多个 HyperLogLog 时，复杂度为 O(N) ，并且常数时间也比处理单个 HyperLogLog 时要大得多。

2.7.3 合并多个 HyperLogLog

```
PFMERGE destkey sourcekey [sourcekey ...]
```

将多个 HyperLogLog 合并为一个 HyperLogLog ，合并后的 HyperLogLog 的基数估算值是通过对所有给定 HyperLogLog 进行并集计算得出的。 命令的复杂度为 O(N) ， 其中 N 为被合并的 HyperLogLog 数量， 不过这个命令的常数复杂度比较高。

3. 专题功能

3.1 排序

redis支持对list，set和sorted set元素的排序。排序命令是sort 完整的命令格式如下：

```
SORT key [BY pattern] [LIMIT start count] [GET pattern] [ASC|DESC] [ALPHA] [STORE dstk
ey]
```

复杂度为O(N+M*log(M))。(N是集合大小，M 为返回元素的数量)

说明：

1. [ASC|DESC] [ALPHA]: sort默认的排序方式（asc）是从小到大排的,当然也可以按照逆序或者按字符顺序排。
2. [BY pattern] : 除了可以按集合元素自身值排序外，还可以将集合元素内容按照给定pattern组合成新的key，并按照新key中对应的内容进行排序。例如：
3. 127.0.0.1:6379sort watch:leto by severtity:* desc
4. [GET pattern]：可以通过get选项去获取指定pattern作为新key对应的值，get选项可以有多个。例如：127.0.0.1:6379sort watch:leto by severtity: *get severtity:*。 对于Hash的引用，采用->，例如：sort watch:leto get # get bug:*->priority。
5. [LIMIT start count] 限定返回结果的数量。
6. [STORE dstkey] 把排序结果缓存起来

调用一次 SORT 命令可以给定多个 GET 选项。

```
redis> SADD names "peter" "tom" "jack"
(integer) 3

redis> MSET peter-name "Peter Hanson" jack-name "Jack Sparrow" tom-name "Thomas Edward"
OK

redis> MSET peter-id 10086 jack-id 255255 tom-id 123321
OK

redis> SORT names ALPHA GET # GET *-id GET *-name
1) "jack"
2) "255255"
3) "Jack Sparrow"
4) "peter"
5) "10086"
6) "Peter Hanson"
7) "tom"
8) "123321"
9) "Thomas Edward"
```

当给定 GET # 时，命令会返回被排序的 值本身。

redis> SORT team-member-ids BY *-KPI GET # GET *-name GET *-KPI
通过 KPI 值，对团队中各个成员的 ID 进行排序
然后根据排序结果，依次获取成员的 ID、名字和 KPI 值

redis> SORT names ALPHA DESC GET # GET *-id GET *-name LIMIT 0 5 STORE profiles
对 names 键的值进行文字形式的降序排序，
根据排序后的前五个 值，获取值本身、及其对应的 ID 和名字
然后将获取到的这些信息储存到 profiles 键里面

3.2 事务

用Multi(Start Transaction)、Exec(Commit)、Discard(Rollback)实现。 在事务提交前，不会执行任何指令，只会把它们存到一个队列里，不影响其他客户端的操作。在事务提交时，批量执行所有指令。 一般情况下redis在接受到一个client发来的命令后会立即处理并返回处理结果，但是当一个client在一个连接中发出multi命令后，这个连接会进入一个事务上下文，该连接后续的命令并不是立即执行，而是先放到一个队列中。当从此连接受到exec命令后，redis会顺序的执行队列中的所有命令。并将所有命令的运行结果打包到一起返回给client.然后此连接就结束事务上下文。

Redis还提供了一个Watch功能，你可以对一个key进行Watch，然后再执行Transactions，在这过程中，如果这个Watched的值进行了修改，那么这个Transactions会发现并拒绝执行。

使用discard命令来取消一个事务。

注意：redis只能保证事务的每个命令连续执行（因为是单线程架构，在执行完事务内所有指令前是不可能再去同时执行其他客户端的请求的，也因此就不存在"事务内的查询要看到事务里的更新，在事务外查询不能看到"这个让人万分头痛的问题），但是如果事务中的一个命令失败了，并不回滚其他命令。另外，一个十分罕见的问题是当事务的执行过程中，如果redis意外的挂了。只有部分命令执行了，后面的也就被丢弃了。注意，如果是笔误，语法出现错误，则整个事务都无法执行。

一个简单案例表明出错也不会回滚：

```
127.0.0.1:6379> del q1
(integer) 0
127.0.0.1:6379> exists q1
(integer) 0
127.0.0.1:6379> multi
OK
127.0.0.1:6379> rpush q1 bar
QUEUED
127.0.0.1:6379> scard q1
QUEUED
127.0.0.1:6379> exec
1) (integer) 1
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> exists q1
(integer) 1
```

当然如果我们使用的append-only file方式持久化，redis会用单个write操作写入整个事务内容。即是是这种方式还是有可能只部分写入了事务到磁盘。发生部分写入事务的情况下，redis重启时会检测到这种情况，然后失败退出。可以使用redis-check-aof工具进行修复，修

复会删除部分写入的事务内容。修复完后就能够重新启动了。

3.3 流水线

利用流水线（pipeline）的方式从client打包多条命令一起发出，不需要等待单条命令的响应返回，而redis服务端会处理完多条命令后会将多条命令的处理结果打包到一起返回给客户端：

```
cat data.txt | redis-cli –pipe
```

在选择开源redis开发库时需要着重注意是否支持pipeline，常见的jedis可以支持。

在部署架构是网络多跳的时候需要注意使用pipeline提高处理效率。

3.4 发布订阅

redis作为一个pub/sub server，在订阅者和发布者之间起到了消息路由的功能。订阅者可以通过subscribe和psubscribe命令向redis server订阅自己感兴趣的消息类型，redis将消息类型称为频道(channel)。当发布者通过publish命令向redis server发送特定类型的消息时。订阅该消息类型的全部client都会收到此消息。这里消息的传递是多对多的。一个client可以订阅多个channel,也可以向多个channel发送消息。

```
终端A：
127.0.0.1:6379> subscribe comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
终端B:
127.0.0.1:6379> subscribe comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
终端C：
127.0.0.1:6379> publish comments good!
(integer) 2
终端A:
127.0.0.1:6379> subscribe comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
1) "message"
2) "comments"
3) "good!"
终端B：
127.0.0.1:6379> subscribe comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
1) "message"
2) "comments"
3) "good!"
```

可以使用psubscribe通过通配符进行多个channel的订阅

4. 开发设计规范

4.1 Key设计

key的一个格式约定：`object-type:id:field`。用":"分隔域，用"."作为单词间的连接，如"`comment:12345:reply.to`"。不推荐含义不清的key和特别长的key。

一般的设计方法如下： 1: 把表名转换为key前缀 如, tag: 2: 第2段放置用于区分区key的字段--对应mysql中的主键的列名,如userid 3: 第3段放置主键值,如2,3,4...., a , b ,c 4: 第4段,写要存储的列名

例如用户表 user, 转换为key-value存储：

userid	username	password	email
9	lisi	1111111	lisi@163.com

```
set user:userid:9:username lisi
set user:userid:9:password 111111
set user:userid:9:email   lisi@163.com
```

例如，查看某个用户的所有信息为：

```
keys user:userid:9*
```

如果另一个列也常常被用来查找，比如username，则也要相应的生成一条按照该列为主的key-value，例如：

```
user:username:lisi:uid 9
```

此时相当于RDBMS中在username上加索引，我们可以根据 `username:lisi:uid` ,查出userid=9，再查`user:9:password/email` ...

4.2 超时设置

从业务需求逻辑和内存的角度，尽可能的设置key存活时间。

4.3 数据异常处理

程序应该处理如果redis数据丢失时的清理redis内存和重新加载的过程。

4.4 内存考虑

1. 只要有可能的话，就尽量使用散列键而不是字符串键来储存键值对数据，因为散列键管理方便、能够避免键名冲突、并且还能够节约内存。

   具体实例： 节约内存：Instagram的Redis实践 blog.nosqlfan.com/html/3379.html

2. 如果将redis作为cache进行频繁读写和超时删除等，此时应该避免设置较大的k-v，因为这样会导致redis的 内存碎片增加，导致rss占用较大，最后被操作系统OOM killer干掉。一个很具体的issue例子请见：https://github.com/antirez/redis/issues/2136

3. 如果采用序列化考虑通用性，请采用json相关的库进行处理，如果对内存大小和速度都很关注的，推荐使用messagepack进行序列化和反序列化

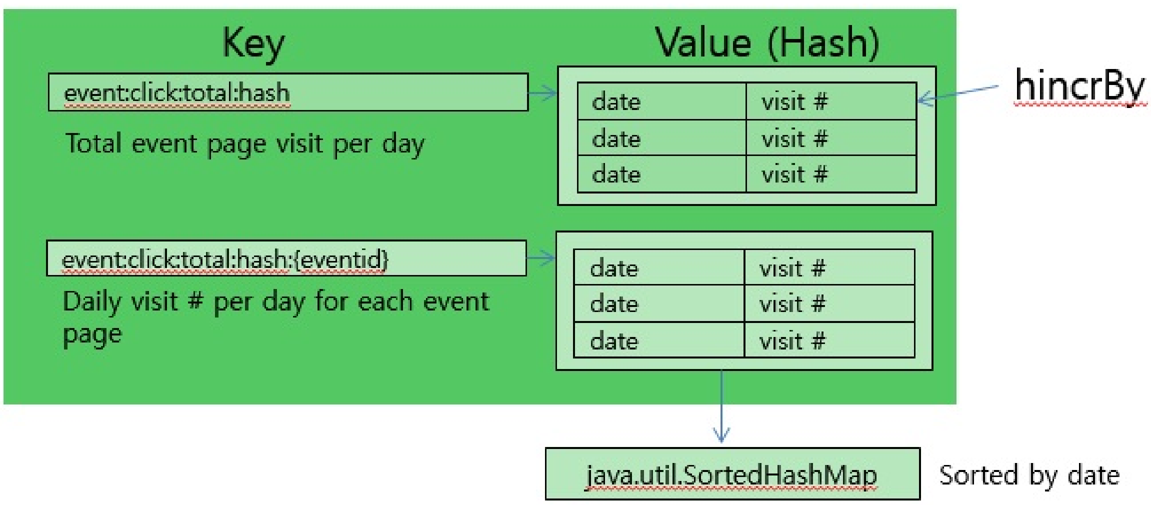
4. 如果需要计数器，请将计数器的key通过天或者小时分割，比如下边的设计:

event:click:total	visit #
event:click:{event page# id}	visit #
event:click:{event page# id}	visit #

   需要修改为：

event:click:daily:total:{date}	visit #
event:click:daily:{date}:{event page# id}	visit #
event:click:daily:{date}:{event page# id}	visit #

   更好的一个设计是采用hash：



5. 各种数据结构及其占用内存的benchmark测试

set个数	每个**set**的元素总数	内存占用	**Key**大小	**Value**大小
100	100	1.88M	7	36
100	1000	10.75M	7	36
100	10000	111.12M	7	36
1000	100	11.59M	8	36
1000	1000	100.35M	8	36
1000	10000	1.08G	8	36
10000	100	108.71M	9	36
10000	1000	996.23M	9	36

zset个数	每个**zset**的元素总数	内存占用	**Key**大小	**Value**大小
100	100	1.62M	8	49
100	1000	15.91M	8	49
100	10000	162.06M	8	49
1000	100	8.71M	9	49
1000	1000	151.87M	9	49
1000	10000	1.58G	9	49
10000	100	79.83M	10	49
10000	1000	1.48G	10	49

hash个数	每个**hash**的元素总数	内存占用	**Key**大小	**Value**大小
100	100	1.63M	8	49
100	1000	6.29M	8	49
100	10000	156.91M	8	49
1000	100	8.71M	9	49
1000	1000	55.59M	9	49
1000	10000	1.52G	9	49
10000	100	79.83M	10	49
10000	1000	548.58M	10	49

list个数	每个list的元素总数	内存占用	Key大小	Value大小
100	100	1.23M	8	36
100	1000	10.00M	8	36
100	10000	92.40M	8	36
1000	100	4.83M	9	36
1000	1000	92.52M	9	36
1000	10000	916.47M	9	36
10000	100	40.76M	10	36
10000	1000	917.69M	10	36

string个数	内存占用	Key大小	Value大小
100	846.79K	13	36
1000	966.29K	13	36
10000	2.16M	13	36
100000	130.88M	13	36

4.5 延迟考虑

1. 尽可能使用批量操作：

- mget、hmget而不是get和hget，对于set也是如此。
- lpush向一个list一次性导入多个元素，而不用lset一个个添加
- LRANGE 一次取出一个范围的元素，也不用LINDEX一个个取出

2. 尽可能的把redis和APP SERVER部署在一个网段甚至一台机器。

3. 对于数据量较大的集合，不要轻易进行删除操作，这样会阻塞服务器，一般采用重命名+批量删除的策略：

排序集合：

```
# Rename the key
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.zset.key", newkey)

# Delete members from the sorted set in batche of 100s
while redis.ZCARD(newkey) > 0
  redis.ZREMRANGEBYRANK(newkey, 0, 99)
end
```

集合：

```
# Rename the key
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.set.key", newkey)

# Delete members from the set in batches of 100
cursor = 0
loop
  cursor, members = redis.SSCAN(newkey, cursor, "COUNT", 100)
  if size of members > 0
redis.SREM(newkey, members)
  end
  if cursor == 0
break
  end
end
```

列表：

```
# Rename the key
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.list.key", newkey)

# Trim off elements in batche of 100s
while redis.LLEN(newkey) > 0
  redis.LTRIM(newkey, 0, -99)
end
```

Hash：

```
# Rename the key
newkey = "gc:hashes:" + redis.INCR( "gc:index" )
redis.RENAME("my.hash.key", newkey)

# Delete fields from the hash in batche of 100s
cursor = 0
loop
  cursor, hash_keys = redis.HSCAN(newkey, cursor, "COUNT", 100)
  if hash_keys count > 0
redis.HDEL(newkey, hash_keys)
  end
  if cursor == 0
break
  end
end
```

4. 尽可能使用不要超过1M大小的kv。

5. 减少对大数据集的高时间复杂度的操作：根据复杂度计算，如下命令可以优化:

命令	时间复杂度	描述	提升性能的建议
ZINTERSTORE	O(N*K)+O(M*log(M))	统计多个有序集合的交集，并存储结果	减少集合的和(或)结果的数量
SINTERSTORE	O(N * M)	统计多个集合的交集，并存储结果	尽可能减少集合的数量(大小)
SINTER	O(N * M)	返回多个集合的交集	尽可能减少集合的数量(大小)
MIGRATE	O(N)	从一个实例上传输key到另一个	减少对象值的数量和(或)大小
DUMP	O(N*M)	返回指定key的序列化结果	减少对象值的数量和(或)大小
ZREM	O(M*log(N))	从有序集合中移除一个或多个成员	减少移除成员的数量和(或)有序集合的大小
ZUNIONSTORE	O(N)+O(M log(M))	统计多个有序集合并集，并存储结果	减少总求并集的集合的总数量和(或)减少结果的数量
SORT	O(N+M*log(M))	排序list、set、sortedset的成员	减少排序的数量和(或)返回成员的数量
SDIFFSTORE	O(N)	统计多个set的差集并存储结果	减少总求差集的集合的数量
SDIFF	O(N)	返回多个set的差集	减少总求差集的集合的数量
SUNION	O(N)	统计多个set的并集	减少总求并集的集合的数量
LSET	O(N)	设置list中某个成员的值	减少list数据结构的长度
LREM	O(N)	从list中移除一定数量成员	减少list数据结构的长度
LRANGE	O(S+N)	返回list中指定区间的集合	减少偏移量和(或)区间的数量

6. 尽可能使用pipeline操作：一次性的发送命令比一个个发要减少网络延迟和单个处理开销。一个性能测试结果为（注意并不是pipeline越大效率越高，注意最后一个测试结果）：

```
logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50
PING_INLINE: 90155.07 requests per second
PING_BULK: 92302.02 requests per second
SET: 85070.18 requests per second
GET: 86184.61 requests per second

logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 10
PING_INLINE: 558035.69 requests per second
PING_BULK: 668002.69 requests per second
SET: 275027.50 requests per second
GET: 376647.84 requests per second

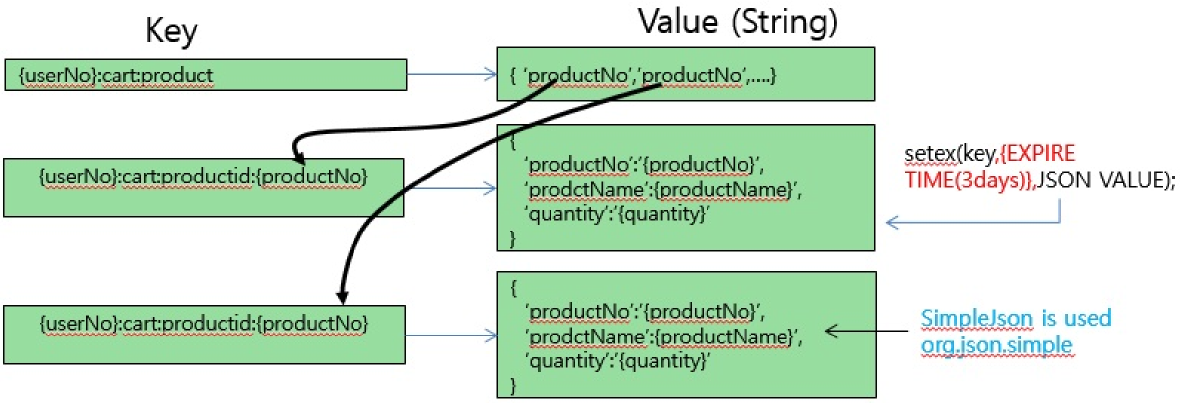
logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 20
PING_INLINE: 705716.25 requests per second
PING_BULK: 869565.25 requests per second
SET: 343406.59 requests per second
GET: 459347.72 requests per second

logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 50
PING_INLINE: 940733.81 requests per second
PING_BULK: 1317523.00 requests per second
SET: 380807.31 requests per second
GET: 523834.47 requests per second

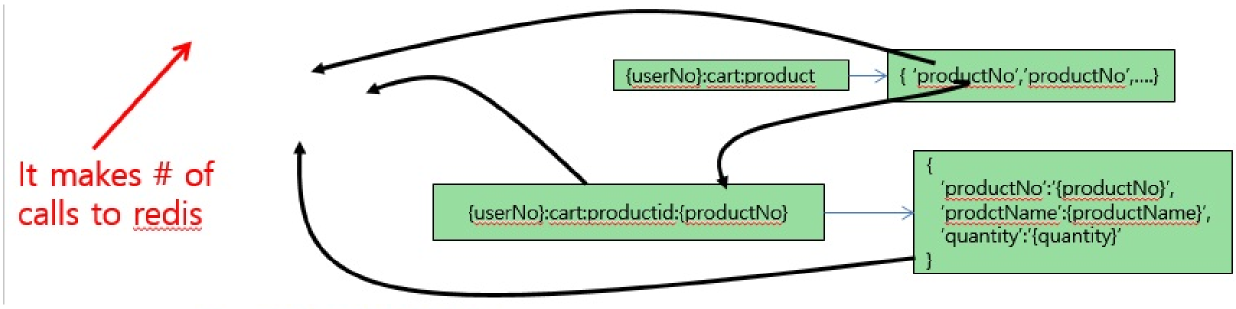
logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 100
PING_INLINE: 999000.94 requests per second
PING_BULK: 1440922.12 requests per second
SET: 386996.88 requests per second
GET: 602046.94 requests per second

logger@BIGD1TMP:~> redis-benchmark -q -r 100000 -n 1000000 -c 50 -P 200
PING_INLINE: 1078748.62 requests per second
PING_BULK: 1381215.50 requests per second
SET: 379218.81 requests per second
GET: 537634.38 requests per second
```

一个场景是一个购物车的设计，一般的设计思路是：



在获取购物车内部货品时，不使用pipeline会很低效：



可以修改为：

```
p = redis.pipelined()
getProductList()
for(productsNo){
  p.get(product)
}
List<Object> redisResult = p.syncAndReturnAll();
for(item:redisResult){
   json.add(item)
}
```

7. 如果出现频繁对**string**进行**append**操作，则请使用**list**进行**push**操作，取出时使用**pop**。这样避免**string**频繁分配内存导致的延时。

8. 如果要**sort**的集合非常大的话排序就会消耗很长时间。由于**redis**单线程的，所以长时间的排序操作会阻塞其他**client**的 请求。解决办法是通过主从复制机制将数据复制到多个**slave**上。然后我们只在**slave**上做排序操作。把可能的对排序结果缓存。另外就是一个方案是就是采用**sorted set**对需要按某个顺序访问的集合建立索引。

4.6 典型使用场景参考

下面是Redis适用的一些场景:

1. 取最新 N 个数据的操作

比如典型的取你网站的最新文章，通过下面方式，我们可以将最新的 5000条评论的ID放在Redis的List集合中，并将超出集合部分从数据库获取。

使用LPUSH latest.comments命令，向 list集合中插入数据 插入完成后再用 LTRIM latest.comments 0 5000 命令使其永远只保存最近5000个 ID 然后我们在客户端获取某一页评论时可以用下面的逻辑

```
FUNCTION get_latest_comments(start,num_items):
 id_list = redis.lrange("latest.comments",start,start+num_items-1)
 IF id_list.length < num_items
 id_list = SQL_DB("SELECT ... ORDER BY time LIMIT ...")
 END
 RETURN id_list
END
```

如果你还有不同的筛选维度，比如某个分类的最新 N 条，那么你可以再建一个按此分类的List，只存ID的话，Redis是非常高效的。

2. 排行榜应用，取 TOP N 操作

这个需求与上面需求的不同之处在于，前面操作以时间为权重，这个是以某个条件为权重，比如按顶的次数排序，这时候就需要我们的 sorted set出马了，将你要排序的值设置成 sorted set的score，将具体的数据设置成相应的 value，每次只需要执行一条ZADD命令即可。

```
127.0.0.1:6379> zdd topapp 1 weixin
(error) ERR unknown command 'zdd'
127.0.0.1:6379> zadd topapp 1 weixin
(integer) 1
127.0.0.1:6379> zadd topapp 1 QQ
(integer) 1
127.0.0.1:6379> zadd topapp 2 meituan
(integer) 1
127.0.0.1:6379> zincrby topapp 1 meituan
"3"
127.0.0.1:6379> zincrby topapp 1 QQ
"2"
127.0.0.1:6379> zrank topapp QQ
(integer) 1
3) "meituan"
127.0.0.1:6379> zrank topapp weixin
(integer) 0
127.0.0.1:6379> zrange topapp 0 -1
1) "weixin"
2) "QQ"
```

3.需要精准设定过期时间的应用

比如你可以把上面说到的 sorted set 的 score 值设置成过期时间的时间戳，那么就可以简单地通过过期时间排序，定时清除过期数据了，不仅是清除 Redis中的过期数据，你完全可以把 Redis 里这个过期时间当成是对数据库中数据的索引，用 Redis 来找出哪些数据需要过期删除，然后再精准地从数据库中删除相应的记录。

4.计数器应用

Redis的命令都是原子性的，你可以轻松地利用 INCR，DECR 命令来构建计数器系统。

5.Uniq 操作，获取某段时间所有数据排重值

这个使用Redis的 set数据结构最合适了，只需要不断地将数据往 set中扔就行了，set意为集合，所以会自动排重。

6.实时系统，反垃圾系统

通过上面说到的 set功能，你可以知道一个终端用户是否进行了某个操作，可以找到其操作的集合并进行分析统计对比等。

7.Pub/Sub 构建实时消息系统

Redis 的 Pub/Sub 系统可以构建实时的消息系统，比如很多用 Pub/Sub 构建的实时聊天系统的例子。

8.构建队列系统

使用list可以构建队列系统，使用 sorted set甚至可以构建有优先级的队列系统。

9.缓存

性能优于Memcached，数据结构更多样化。作为RDBMS的前端挡箭牌，redis可以对一些使用频率极高的sql操作进行cache，比如，我们可以根据sql的hash进行SQL结果的缓存：

```
def get_results(sql):
    hash = md5.new(sql).digest()
    result = redis.get(hash)
    if result is None:
        result = db.execute(sql)
        redis.set(hash, result)
        # or use redis.setex to set a TTL for the key
    return result
```

10.使用setbit进行统计计数

下边的例子是记录UV

```
#!/usr/bin/python
import redis
from bitarray import bitarray
from datetime import date

r=redis.Redis(host='localhost', port=6379, db=0)
today=date.today().strftime('%Y-%m-%d')

def bitcount(n):
len(bin(n)-2)

def setup():
r.delete('user:'+today)
r.setbit('user:'+today,100,0)

def setuniquser(uid):
r.setbit('user:'+today,uid,1)

def countuniquser():
a = bitarray()
a.frombytes(r.get('user:'+today),)
print a
print a.count()


if __name__=='__main__':
setup()
setuniquser(uid=0)
countuniquser()
```

11.维护好友关系

使用set进行是否为好友关系，共同好友等操作

12.使用 Redis 实现自动补全功能

使用有序集合保存输入结果：

```
ZADD word:a 0 apple 0 application 0 acfun 0 adobe
ZADD word:ap 0 apple 0 application
ZADD word:app 0 apple 0 application
ZADD word:appl 0 apple 0 application
ZADD word:apple 0 apple
ZADD word:appli 0 application
```

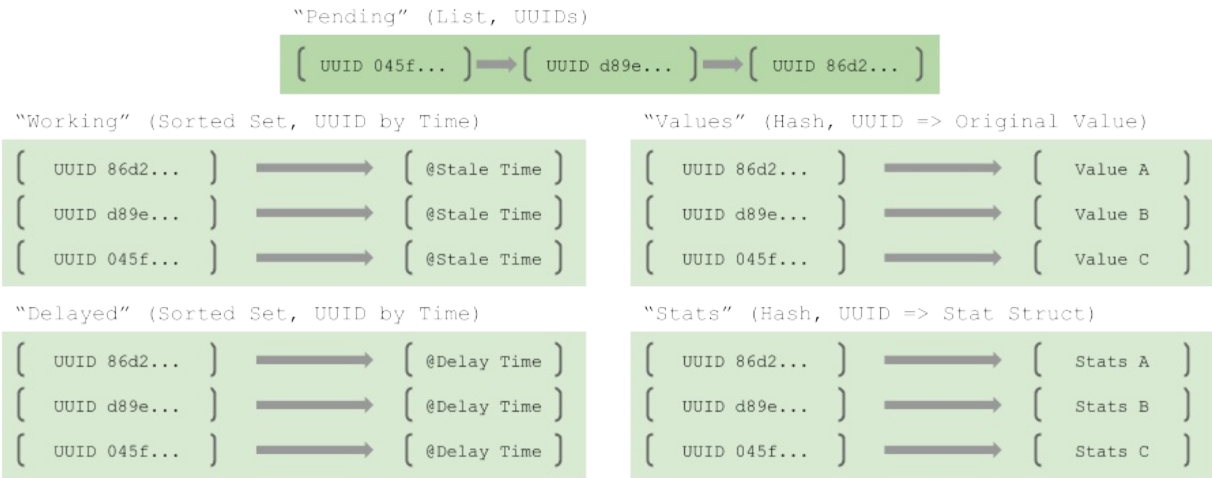
再使用一个有序集合保存热度：

```
ZADD word_scores 100 apple 80 adobe 70 application 60 acfun
```

取结果时采用交集操作：

```
ZINTERSTORE word_result 2 word_scores word:a WEIGHTS 1 1
ZRANGE word_result 0 -1 withscores
```

13. 可靠队列设计



• UUIDs as Surrogate Keys Our strategy spreads information about the state of an item in the queue across a number of Redis data structures, requiring the use of a per-item surrogate key to tie them together. The UUID is a good choice here because 1) they are quick to generate, and 2) can be generated by the clients in a decentralized manner. • Pending List The Pending List holds the generated UUIDs for the items that have been enqueued(), and are ready to be processed. It is a RedisList, presenting the generated UUIDs in FIFO order. • Values Hash The Values Hash holds the actual items that have been enqueued. It is a Redis Hash, mapping the generated UUID to the binary form of the the item. This is the only representation of the original item that will appear in any of the data structures. • Stats Hash The Stats Hash records some timestamps and counts for each of the items. Specifically: • enqueue time • last dequeue time • dequeue count • last requeue time • last requeue count. It is a Redis Hash, mapping the generated UUID to a custom data structure that holds this data in a packed, binary form. Why keep stats on a per-item basis? We find it really useful for debugging (e.g. do we have a bad apple item that is being continuously requeued?), and for understanding how far behind we are if queues start to back up. Furthermore, the cost is only ~40 bytes per item, much smaller than our typical queued items. • Working Set The Working Set holds the set of UUIDs that have been dequeued(), and are currently being processed. It is a Redis Sorted Set, and scores each of the UUIDs by a pre-calculated, millisecond timestamp. Any object that has exceeded its assigned timeout is considered abandoned, and is available to be reclaimed. • Delay Set

The Delay Set holds the set of UUIDs that have been requeued() with a per-item deferment. It is a Redis Sorted Set, and scores each of the UUIDs by a pre-calculated, millisecond timestamp. Once the deferment timestamp has expired, the item will be returned to the Pending List. Why support per-item deferment? We have a number of use cases where we might want to backoff specific pieces of work — maybe an underlying resource is too busy — without backing off the entire queue. Per-item deferment lets us say, “requeue this item, but don’t make it available for dequeue for another n seconds.”

4.7 客户端推荐

4.7.1 Redis-Python驱动的安装和使用

```
unzip redis-py-master.zip
cd redis-py-master/
sudo python setup.py install
```

完成后import redis即可。

4.7.2 Redis-Java客户端推荐

1. Jedis ：https://github.com/xetorthio/jedis 重点推荐
2. Spring Data redis ：https://github.com/spring-projects/spring-data-redis 使用Spring框架时推荐
3. Redisson ：https://github.com/mrniko/redisson 分布式锁、阻塞队列的时重点推荐

4.7.3 Redis-C客户端推荐

Hiredis是redis数据库一个官方推荐的C语言redis client库。

5. 上线部署规划

5.1 内存、CPU规划

一定要设置最大内存maxmemory参数，否则物理内存用爆了就会大量使用Swap，写RDB文件时的速度很慢。注意这个参数指的是info中的used_memory，在一些不利于jmalloc的时候，内存碎片会很大。

多留55%内存是最安全的。重写AOF文件和RDB文件的进程(即使不做持久化，复制到Slave的时候也要写RDB)会fork出一条新进程来，采用了操作系统的Copy-On-Write策略(子进程与父进程共享Page。如果父进程的Page-每页4K有修改，父进程自己创建那个Page的副本，不会影响到子进程)。

另外，需要考虑内存碎片，假设碎片为1.2，则如果机器为64G，那么64*45%/1.2 = 24G作为maxmemory是比较安全的规划。

留意Console打出来的报告，如"RDB: 1215 MB of memory used by copy-on-write"。在系统极度繁忙时，如果父进程的所有Page在子进程写RDB过程中都被修改过了，就需要两倍内存。

按照Redis启动时的提醒，设置

```
echo "vm.overcommit_memory = 1" >>  /etc/sysctl.conf
```

使得fork()一条10G的进程时，因为COW策略而不一定需要有10G的free memory。

另外，记得关闭THP，这个默认的Linux内存页面大小分配策略会导致RDB时出现巨大的latency和巨大的内存占用。关闭方法为：

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

当最大内存到达时，按照配置的Policy进行处理， 默认策略为volatile-lru，对设置了expire time的key进行LRU清除(不是按实际expire time)。如果没有数据设置了expire time或者policy为noeviction，则直接报错，但此时系统仍支持get之类的读操作。 另外还有几种policy，比如volatile-ttl按最接近expire time的，allkeys-lru对所有key都做LRU。注意在一般的缓存系统中，如果没有设置超时时间，则lru的策略需要设置为allkeys-lru，并且应用需要做好未命中的异常处理。特殊的，当redis当做DB时，请使用noneviction策略，但是需要对系统内存监控加强粒度。

CPU不求核数多，但求主频高，Cache大，因为redis主处理模式是单进程的。同时避免使用虚拟机。

5.2 网卡RPS设置

RPS就是让网卡使用多核CPU的。传统方法就是网卡多队列（RSS，需要硬件和驱动支持），RPS则是在系统层实现了分发和均衡。如果对redis网络处理能力要求高或者在生产上发现cpu0的，可以在OS层面打开这个内核功能。

设置脚本：

```
#!/bin/bash
# Enable RPS (Receive Packet Steering)

rfc=32768
cc=$(grep -c processor /proc/cpuinfo)
rsfe=$(echo $cc*$rfc | bc)
sysctl -w net.core.rps_sock_flow_entries=$rsfe
for fileRps in $(ls /sys/class/net/eth*/queues/rx-*/rps_cpus)
do
echo fff > $fileRps
done

for fileRfc in $(ls /sys/class/net/eth*/queues/rx-*/rps_flow_cnt)
do
echo $rfc > $fileRfc
done

tail /sys/class/net/eth*/queues/rx-*/{rps_cpus,rps_flow_cnt}
```

5.3 服务器部署位置

尽可能把client和server部署在同一台机器上，比如都部署在app server，或者一个网段中，减少网络延迟对于redis的影响。

如果是同一台机器，又想榨干redis性能可以考虑采用UNIX domain sockets配置方式，配置方式如下

```
# 0 = do not listen on a port
port 0

# listen on localhost only
bind 127.0.0.1

# create a unix domain socket to listen on
unixsocket /tmp/redis.sock

# set permissions for the socket
unixsocketperm 755
```

这样的配置方式在没有大量pipeline下会有一定性能提升，具体请参见 http://redis.io/topics/benchmarks：

另外，对于混合部署即redis和应用部署在同一台服务器上，那么可能会出现如下的情况：

> 出现瞬时 Redis 大量连接和处理超时，应用业务线程被阻塞，导致服务拒绝，过一段时间可能又自动恢复了。这种瞬时故障非常难抓现场，一天来上几发就会给人业务不稳定的感受，而一般基础机器指标的监控周期在分钟级。瞬时故障可能发生在监控的采集间隙，所以只好上脚本在秒级监控日志，发现瞬时出现大量 Redis 超时错误，就收集当时应用的 JVM 堆栈、内存和机器 CPU Load 等各项指标。终于发现瞬时故障时刻 Redis 机器 CPU Load 出现瞬间飙升几百的现象，应用和 Redis 混合部署时应用可能瞬间抢占了全部 CPU 导致 Redis 没有 CPU 资源可用。而应用处理业务的逻辑又可能需要访问 Redis，而 Redis 又没有 CPU 资源可用导致超时，这不就像一个死锁么。搞清楚了原因其实解决方法也简单，就是分离应用和 Redis 的部署，各自资源隔离
>
> 出处： http://mp.weixin.qq.com/s?__biz=MzAxMTEyOTQ5OQ==&mid=402004912&idx=1&sn=7517696a86f54262e60e1b5636d6cbe0&3rd=MzA3MDU4NTYzMw==&scene=6#rd

因此在混合部署下要对极限性能进行监控，提前将可能出现性能问题的应用迁移出来。

5.4 持久化设置

RDB和AOF两者毫无关系，完全独立运行，如果使用了AOF，重启时只会从AOF文件载入数据，不会再管RDB文件。在配置上有三种选择：不持久化，RDB，RDB+AOF。官方不推荐只开启AOF（因为恢复太慢另外如果aof引擎有bug），除非明显的读多写少的应用。 开启AOF时应当关闭AOF自动rewrite，并在crontab中启动在业务低峰时段进行的bgrewrite。 如果在一台机器上部署多个redis实例，则关闭RDB和AOF的自动保存（save "", auto-aof-rewrite-percentage 0），通过crontab定时调用保存：

```
m h * * * redis-cli -p <port> BGSAVE
m h */4 * * redis-cli -p <port> BGREWRITEAOF
```

持久化的部署规划上，如果为主从复制关系，建议主关闭持久化。

5.5 多实例配置

如果一台机器上防止多个redis实例，为了防止上下文切换导致的开销，可以采用taskset。taskset是LINUX提供的一个命令(ubuntu系统可能需要自行安装，schedutils package)。他可以让某个程序运行在某个（或）某些CPU上。

1）显示进程运行的CPU （6137为redis-server的进程号）

```
[redis@hadoop1 ~]$ taskset  -p 6137
pid 6137's current affinity mask: f
```

显示结果的f实际上是二进制4个低位均为1的bitmask，每一个1对应于1个CPU，表示该进程在4个CPU上运行

2）指定进程运行在某个特定的CPU上

```
[redis@hadoop1 ~]$ taskset -pc 3 6137
pid 6137's current affinity list: 0-3
pid 6137's new affinity list: 3
```

注：3表示CPU将只会运行在第4个CPU上（从0开始计数）。

3）进程启动时指定CPU

```
taskset -c 1 ./redis-server ../redis.conf
```

参数：OPTIONS -p, --pid operate on an existing PID and not launch a new task

-c, --cpu-list specify a numerical list of processors instead of a bitmask. The list may contain multiple items, separated by comma, and ranges. For example, 0,5,7,9-11.

5.6 具体设置参数

详见我行自制安装包conf目录中的各个配置文件和上线前检查表格。

redis参数设置技巧列表：

1. Daemonize 这个参数在使用supervisord这种进程管理工具时一定要设置为no，否则无法使用这些工具将redis启动。
2. Dir RDB的位置，一定要事先创建好，并且启动redis 的用户对此目录要有读写权限。
3. Include 如果是多实例的话可以将公共的设置放在一个conf文件中，然后引用即可：include /redis/conf/redis-common.conf

5.7 其他好用的配置技巧

5.7.1 使用supervisord进行进程管理

Supervisord是一个优秀的进程管理工具，一般在部署redis时采用它来进行redis、sentinel等进程的管理，一个已经在生产环境采用的supervisord配置文件如下： ; Sample supervisor config file. ; ; For more information on the config file, please see: ; http://supervisord.org/configuration.html ; ; Notes: ; - Shell expansion ("~" or "$HOME") is not supported. Environment ; variables can be expanded using this syntax: "%(ENV_HOME)s". ; - Comments must have a leading space: "a=b ;comment" not "a=b;comment".

```
[unix_http_server]
file=/smsred/redis-3.0.4/run/supervisor.sock ; (the path to the socket file)
;chmod=0700 ; socket file mode (default 0700)
;chown=nobody:nogroup ; socket file uid:gid owner
;username=user ; (default is no username (open server))
;password=123 ; (default is no password (open server))

;[inet_http_server] ; inet (TCP) server disabled by default
;port=127.0.0.1:9001 ; (ip_address:port specifier, *:port for all iface)
;username=user ; (default is no username (open server))
;password=123 ; (default is no password (open server))

[supervisord]
logfile=/smsred/redis-3.0.4/log/supervisord.log ; (main log file;default $CWD/supervis
ord.log)
logfile_maxbytes=50MB ; (max main logfile bytes b4 rotation;default 50MB)
logfile_backups=10 ; (num of main logfile rotation backups;default 10)
loglevel=info ; (log level;default info; others: debug,warn,trace)
pidfile=/smsred/redis-3.0.4/run/supervisord.pid ; (supervisord pidfile;default supervi
sord.pid)
nodaemon=false ; (start in foreground if true;default false)
minfds=1024 ; (min. avail startup file descriptors;default 1024)
minprocs=200 ; (min. avail process descriptors;default 200)
;umask=022 ; (process file creation umask;default 022)
;user=chrism ; (default is current user, required if root)
;identifier=supervisor ; (supervisord identifier, default is 'supervisor')
;directory=/tmp ; (default is not to cd during start)
;nocleanup=true ; (don't clean up tempfiles at start;default false)
;childlogdir=/tmp ; ('AUTO' child log dir, default $TEMP)
;environment=KEY="value" ; (key value pairs to add to environment)
;strip_ansi=false ; (strip ansi escape codes in logs; def. false)

; the below section must remain in the config file for RPC
; (supervisorctl/web interface) to work, additional interfaces may be
; added by defining them in separate rpcinterface: sections
```

```
[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=unix:///smsred/redis-3.0.4/run/supervisor.sock ; use a unix:// URL for a uni
x socket
;serverurl=http://127.0.0.1:9001 ; use an http:// url to specify an inet socket
;username=chris ; should be same as http_username if set
;password=123 ; should be same as http_password if set
;prompt=mysupervisor ; cmd line prompt (default "supervisor")
;history_file=~/.sc_history ; use readline history if available

; The below sample program section shows all possible program subsection values,
; create one or more 'real' program: sections to be able to control them under
; supervisor.

;[program:theprogramname]
;command=/bin/cat ; the program (relative uses PATH, can take args)
;process_name=%(program_name)s ; process_name expr (default %(program_name)s)
;numprocs=1 ; number of processes copies to start (def 1)
;directory=/tmp ; directory to cwd to before exec (def no cwd)
;umask=022 ; umask for process (default None)
;priority=999 ; the relative start priority (default 999)
;autostart=true ; start at supervisord start (default: true)
;autorestart=unexpected ; whether/when to restart (default: unexpected)
;startsecs=1 ; number of secs prog must stay running (def. 1)
;startretries=3 ; max # of serial start failures (default 3)
;exitcodes=0,2 ; 'expected' exit codes for process (default 0,2)
;stopsignal=QUIT ; signal used to kill process (default TERM)
;stopwaitsecs=10 ; max num secs to wait b4 SIGKILL (default 10)
;stopasgroup=false ; send stop signal to the UNIX process group (default false)
;killasgroup=false ; SIGKILL the UNIX process group (def false)
;user=chrism ; setuid to this UNIX account to run the program
;redirect_stderr=true ; redirect proc stderr to stdout (default false)
;stdout_logfile=/a/path ; stdout log path, NONE for none; default AUTO
;stdout_logfile_maxbytes=1MB ; max # logfile bytes b4 rotation (default 50MB)
;stdout_logfile_backups=10 ; # of stdout logfile backups (default 10)
;stdout_capture_maxbytes=1MB ; number of bytes in 'capturemode' (default 0)
;stdout_events_enabled=false ; emit events on stdout writes (default false)
;stderr_logfile=/a/path ; stderr log path, NONE for none; default AUTO
;stderr_logfile_maxbytes=1MB ; max # logfile bytes b4 rotation (default 50MB)
;stderr_logfile_backups=10 ; # of stderr logfile backups (default 10)
;stderr_capture_maxbytes=1MB ; number of bytes in 'capturemode' (default 0)
;stderr_events_enabled=false ; emit events on stderr writes (default false)
;environment=A="1",B="2" ; process environment additions (def no adds)
;serverurl=AUTO ; override serverurl computation (childutils)

; The below sample eventlistener section shows all possible
; eventlistener subsection values, create one or more 'real'
; eventlistener: sections to be able to handle event notifications
; sent by supervisor.

;[eventlistener:theeventlistenername]
```

```
;command=/bin/eventlistener ; the program (relative uses PATH, can take args)
;process_name=%(program_name)s ; process_name expr (default %(program_name)s)
;numprocs=1 ; number of processes copies to start (def 1)
;events=EVENT ; event notif. types to subscribe to (req'd)
;buffer_size=10 ; event buffer queue size (default 10)
;directory=/tmp ; directory to cwd to before exec (def no cwd)
;umask=022 ; umask for process (default None)
;priority=-1 ; the relative start priority (default -1)
;autostart=true ; start at supervisord start (default: true)
;autorestart=unexpected ; whether/when to restart (default: unexpected)
;startsecs=1 ; number of secs prog must stay running (def. 1)
;startretries=3 ; max # of serial start failures (default 3)
;exitcodes=0,2 ; 'expected' exit codes for process (default 0,2)
;stopsignal=QUIT ; signal used to kill process (default TERM)
;stopwaitsecs=10 ; max num secs to wait b4 SIGKILL (default 10)
;stopasgroup=false ; send stop signal to the UNIX process group (default false)
;killasgroup=false ; SIGKILL the UNIX process group (def false)
;user=chrism ; setuid to this UNIX account to run the program
;redirect_stderr=true ; redirect proc stderr to stdout (default false)
;stdout_logfile=/a/path ; stdout log path, NONE for none; default AUTO
;stdout_logfile_maxbytes=1MB ; max # logfile bytes b4 rotation (default 50MB)
;stdout_logfile_backups=10 ; # of stdout logfile backups (default 10)
;stdout_events_enabled=false ; emit events on stdout writes (default false)
;stderr_logfile=/a/path ; stderr log path, NONE for none; default AUTO
;stderr_logfile_maxbytes=1MB ; max # logfile bytes b4 rotation (default 50MB)
;stderr_logfile_backups ; # of stderr logfile backups (default 10)
;stderr_events_enabled=false ; emit events on stderr writes (default false)
;environment=A="1",B="2" ; process environment additions
;serverurl=AUTO ; override serverurl computation (childutils)

; The below sample group section shows all possible group values,
; create one or more 'real' group: sections to create "heterogeneous"
; process groups.

;[group:thegroupname]
;programs=progname1,progname2 ; each refers to 'x' in [program:x] definitions
;priority=999 ; the relative start priority (default 999)

; The [include] section can just contain the "files" setting. This
; setting can list multiple files (separated by whitespace or
; newlines). It can also contain wildcards. The filenames are
; interpreted as relative to this file. Included files *cannot*
; include files themselves.

;[include]
;files = relative/directory/*.ini

[program:redis-1xxx]
command=/smsred/redis-3.0.4/bin/redis-server /smsred/redis-3.0.4/conf/redis-1xxx.conf
autostart=true
autorestart=false
user=smsred
stdout_logfile=/smsred/redis-3.0.4/log/redis-1xxx.out.log
```

```
stderr_logfile=/smsred/redis-3.0.4/log/redis-1xxx.err.log
priority=1000

[program:redis-1xxx]
command=/smsred/redis-3.0.4/bin/redis-server /smsred/redis-3.0.4/conf/redis-1xxx.conf
autostart=true
autorestart=false
user=smsred
stdout_logfile=/smsred/redis-3.0.4/log/redis-1xxx.out.log
stderr_logfile=/smsred/redis-3.0.4/log/redis-1xxx.err.log
priority=1000

[program:redis-1xxx]
command=/smsred/redis-3.0.4/bin/redis-server /smsred/redis-3.0.4/conf/redis-1xxx.conf
autostart=true
autorestart=false
user=smsred
stdout_logfile=/smsred/redis-3.0.4/log/redis-1xxx.out.log
stderr_logfile=/smsred/redis-3.0.4/log/redis-1xxx.err.log
priority=1000

[program:redis-sentinel]
command =/smsred/redis-3.0.4/bin/redis-sentinel /smsred/redis-3.0.4/conf/sentinel.conf

autostart=true
autorestart=true
startsecs=3
```

5.7.2 使用alias方便操作

如果开多实例，那么shell下进行操作的次数会很多，因此你需要一些alias进行命令的缩短，这个技巧并不高深，但是很实用。一个实例如下：

```
alias cli1='$HOME/redis-3.0.4/bin/redis-cli -a xxx -p 1xx'
alias cli2='$HOME/redis-3.0.4/bin/redis-cli -a xxx -p 1xx'
alias cli3='$HOME/redis-3.0.4/bin/redis-cli -a xxx -p 1xx'
alias clis='$HOME/redis-3.0.4/bin/redis-cli -p 26379'

alias sctl='supervisorctl -c $HOME/redis-3.0.4/conf/redis-supervisord.conf '
alias sstart='supervisord -c $HOME/redis-3.0.4/conf/redis-supervisord.conf'
alias see='pdsh -R ssh -w MSMSRED[1-3],PSMSRED1,PSMSAPP1 "/usr/local/bin/supervisorctl
 -c /smsred/redis-3.0.4/conf/redis-supervisord.conf status "'
```

5.7.3 使用pdsh/pdcp进行多机器操作

Pdsh/pdcp是一个python ssh多机操作的工具，在部署中可以采用它进行多机的同一操作批量执行，注意编译的时候把ssh编译进去，在执行时指定ssh模式，一个查看多机supervisord管理进程的命令实例如下：

```
pdsh -R ssh -w MSMSRED[1-3],PSMSRED1,PSMSAPP1 "/usr/local/bin/supervisorctl -c /smsred
/redis-3.0.4/conf/redis-supervisord.conf status "
```

前提是你这些机器已经建立了ssh互信。建立互信可以用下边这个脚本

```
#!/bin/bash
#2015-12-08
#author gnuhpc

expect -c "spawn ssh-keygen -t rsa
expect {
\":\" {send \"\r\"; exp_continue}
\"*(y/n)*\" {send \"y\r\"; exp_continue}
}
"
for p in $(cat ip.cfg)
do
ip=$(echo "$p"|cut -f1 -d":")
username=$(echo "$p"|cut -f2 -d":")
password=$(echo "$p"|cut -f3 -d":")
echo $password

expect -c "
spawn ssh-copy-id ${username}@$ip
expect {
\"*yes/no*\" {send \"yes\r\"; exp_continue}
\"*(y/n)*\" {send \"y\r\"; exp_continue}
\"*password*\" {send \"$password\r\"; exp_continue}
\"*Password*\" {send \"$password\r\"; exp_continue}
}
"
expect -c "
spawn ssh ${username}@$ip "hostname"
expect {
\"*yes/no*\" {send \"yes\r\"; exp_continue}
\"*password*\" {send \"$password\r\"; exp_continue}
\"*(y/n)*\" {send \"y\r\"; exp_continue}
\"*Password*\" {send \"$password\r\";}
}
"
done
```

指定一个ip.cfg，里面的格式为：IP（主机名也行，只要能解析）:用户名:密码 例如：

```
xxxx.139:username:password
xxxx.140:username:password
xxxx.141:username:password
xxxx.142:username:password
xxxx.137:username:password
```

5.7.4 使用脚本进行sentinel配置文件的备份

Sentinel在启动、切换时会对config文件进行rewrite，在上线前或者某些手动维护后你可能希望把conf文件都变为最初，当系统中有很多redis实例时，这个手工操作会让人疯掉，那不妨写个脚本在配置好sentinel和redis后不启动先备份一下，测试完毕后再恢复。

一个简单的备份脚本如下：

```
backupconf.sh
#!/bin/bash
for i in `find ~/redis-3.0.4/conf -name *.conf`
do
cp -v $i ${i}.bak
done
```

恢复脚本：

```
 recoveryconf.sh
#!/bin/bash
for i in `find ~/redis-3.0.4/conf -name *.conf.bak`
do
cp -v $i ${i%.*}
done
```

6. 常见运维操作

3.1 启动

3.1.1 启动redis

```
$ redis-server redis.conf
```

常见选项： ./redis-server (run the server with default conf) ./redis-server /etc/redis/6379.conf ./redis-server --port 7777 ./redis-server --port 7777 --slaveof 127.0.0.1 8888 ./redis-server /etc/myredis.conf --loglevel verbose

3.1.2 启动redis-sentinel

```
    ./redis-server /etc/sentinel.conf –sentinel
    ./redis-sentinel /etc/sentinel.conf
```

部署后可以使用sstart对redis 和sentinel进行拉起，使用sctl进行supervisorctl的控制。（两个alias）

3.2 停止

```
redis-cli shutdown
```

sentinel方法一样，只是需要执行sentinel的连接端口

> 注意：正确关闭服务器方式是redis-cli shutdown 或者 kill，都会graceful shutdown，保证写RDB文件以及将AOF文件fsync到磁盘，不会丢失数据。 如果是粗暴的Ctrl+C，或者kill -9 就可能丢失。如果有配置save，还希望在shutdown时进行RDB写入，那么请使用shutdown save命令。

3.3 查看和修改配置

查看：

```
config get ：获取服务器配置信息。
redis 127.0.0.1:6379> config get dir
config get *：查看所有配置
```

修改：

```
临时设置：config set
永久设置：config rewrite，将目前服务器的参数配置写入redis conf.
```

3.4 批量执行操作

使用telnet也可以连接redis-server。并且在脚本中使用nc命令进行redis操作也是很有效的：

```
gnuhpc@gnuhpc:~$ (echo -en "ping\r\nset key abc\r\nget key\r\n";sleep 1) | nc 127.0.0.
1 6379
+PONG
+OK
$3
abc
```

另一个方式是使用pipeline：

```
在一个脚本中批量执行多个写入操作:
先把插入操作放入操作文本insert.dat：
set a b
set 1 2
set h w
set f u
然后执行命令:cat insert.bat | ./redis-cli --pipe，或者如下脚本：
#!/bin/sh
host=$1
port=$；
password=$3
cat insert.dat | ./redis-cli -h $host -p $port -a $password --pipe
```

3.5 选择数据库

```
select db-index
```

默认连接的数据库所有是0,默认数据库数是16个。返回1表示成功，0失败

3.6 清空数据库

```
flushdb
```

删除当前选择数据库中的所有 key。生产上已经禁止。

```
flushall
```

删除所有的数据库。生产上已经禁止。

3.7 重命名命令

```
rename-command
```

例如：rename-command FLUSHALL ""。必须重启。

3.8 执行lua脚本

```
--eval
```

例如： redis-cli --eval myscript.lua key1 key2 , arg1 arg2 arg3

3.9 设置密码

```
config set requirepass [passw0rd]
```

3.10 验证密码

```
auth passw0rd
```

3.11 性能测试命令

```
redis-benchmark -q -r 100000 -n 100000 -c 50
```

比如：开100条线程(默认50)，SET 1千万次(key在0-1千万间随机)，key长21字节，value长256字节的数据。-r指的是使用随机key的范围。

```
redis-benchmark -t SET -c 100 -n 10000000 -r 10000000 -d 256
```

也可以直接执行lua脚本模拟客户端

```
redis-benchmark -n 100000 -q script load "redis.call('set','foo','bar')"
```

注意：Redis-Benchmark的测试结果提供了一个保证你的 Redis-Server 不会运行在非正常状态下的基准点，但是你永远不要把它作为一个真实的“压力测试”。压力测试需要反应出应用的运行方式，并且需要一个尽可能的和生产相似的环境。

Redis-benchmark还有一个作用就是灌数据，例如下列测试场景，我们对某个系统常用redis API进行测试，下列是一个测试hget、hset的过程，我们首先利用**rand_int**进行随机整数获取，对myhash这个key进行测试数据灌入（这也就测试了hset性能），然后再对其进行hget：

```
MSMSAPP1:/tmp # ./redis-benchmark -a pass -h 40.XXX.XXX.141 -p 16XXXX -r 500000 -n 500
000 hset myhash __rand_int__ __rand_int__
====== hset myhash __rand_int__ __rand_int__ ======
500000 requests completed in 18.74 seconds
50 parallel clients
3 bytes payload
keep alive: 1

23.53% <= 1 milliseconds
95.84% <= 2 milliseconds
97.62% <= 3 milliseconds
97.71% <= 4 milliseconds
97.80% <= 5 milliseconds
97.84% <= 6 milliseconds
97.84% <= 8 milliseconds
97.85% <= 9 milliseconds
97.85% <= 10 milliseconds
97.85% <= 11 milliseconds
97.86% <= 12 milliseconds
97.88% <= 13 milliseconds
97.90% <= 14 milliseconds
97.92% <= 15 milliseconds
```

```
97.94% <= 16 milliseconds
97.94% <= 20 milliseconds
97.95% <= 21 milliseconds
97.95% <= 22 milliseconds
97.99% <= 23 milliseconds
98.01% <= 24 milliseconds
98.11% <= 25 milliseconds
98.43% <= 26 milliseconds
98.85% <= 27 milliseconds
99.17% <= 28 milliseconds
99.43% <= 29 milliseconds
99.54% <= 30 milliseconds
99.68% <= 31 milliseconds
99.77% <= 32 milliseconds
99.81% <= 33 milliseconds
99.85% <= 34 milliseconds
99.85% <= 35 milliseconds
99.87% <= 36 milliseconds
99.88% <= 37 milliseconds
99.89% <= 38 milliseconds
99.90% <= 39 milliseconds
99.90% <= 40 milliseconds
99.91% <= 44 milliseconds
99.91% <= 45 milliseconds
99.91% <= 46 milliseconds
99.92% <= 47 milliseconds
99.92% <= 48 milliseconds
99.93% <= 49 milliseconds
99.93% <= 50 milliseconds
99.95% <= 51 milliseconds
99.96% <= 52 milliseconds
99.96% <= 53 milliseconds
99.97% <= 54 milliseconds
99.98% <= 55 milliseconds
100.00% <= 55 milliseconds
26679.47 requests per second

MSMSAPP1:/tmp # ./redis-benchmark -a pass 40.XXX.XXX.141 -p 16XXXX -r 500000 -n 500000
 hget myhash __rand_int__ __rand_int__
====== hget myhash __rand_int__ __rand_int__ ======
500000 requests completed in 13.83 seconds
50 parallel clients
3 bytes payload
keep alive: 1

74.29% <= 1 milliseconds
98.29% <= 2 milliseconds
98.45% <= 3 milliseconds
98.45% <= 4 milliseconds
98.45% <= 5 milliseconds
98.46% <= 11 milliseconds
98.46% <= 12 milliseconds
98.48% <= 15 milliseconds
```

```
98.49% <= 16 milliseconds
98.50% <= 22 milliseconds
98.50% <= 23 milliseconds
98.57% <= 24 milliseconds
98.81% <= 25 milliseconds
99.16% <= 26 milliseconds
99.45% <= 27 milliseconds
99.71% <= 28 milliseconds
99.84% <= 29 milliseconds
99.91% <= 30 milliseconds
99.94% <= 31 milliseconds
99.94% <= 32 milliseconds
99.95% <= 33 milliseconds
99.96% <= 34 milliseconds
99.96% <= 44 milliseconds
99.96% <= 45 milliseconds
99.97% <= 49 milliseconds
99.97% <= 50 milliseconds
99.99% <= 55 milliseconds
100.00% <= 56 milliseconds
100.00% <= 56 milliseconds
36145.45 requests per second
```

注意：上述测试由于是取的随机值，因此hget可能没有命中，同时payload比较小，所以这是个极限性能。

另外，还有一个工具是RedisLab放出来的，我并没有进行测试 参见：https://github.com/RedisLabs/memtier_benchmark

3.12 Redis-cli命令行其他操作

1. echo ：在命令行打印一些内容

redis 127.0.0.1:6379> echo HongWan "HongWan"

2. quit ：退出连接。

redis 127.0.0.1:6379> quit

3. -x选项从标准输入（stdin）读取最后一个参数。 比如从管道中读取输入：

echo -en "chen.qun" | redis-cli -x set name

4. -r -i

-r 选项重复执行一个命令指定的次数。 -i 设置命令执行的间隔。 比如查看redis每秒执行的commands（qps） redis-cli -r 100 -i 1 info stats | grep instantaneous_ops_per_sec 这个选项在编写一些脚本时非常有用

5. -c：开启reidis cluster模式，连接redis cluster节点时候使用 。

6. --rdb：获取指定redis实例的rdb文件,保存到本地。

redis-cli -h 192.168.44.16 -p 6379 --rdb 6379.rdb

7. --slave

模拟slave从master上接收到的commands。slave上接收到的commands都是update操作，记录数据的更新行为。

8. --pipe

这个一个非常有用的参数。发送原始的redis protocl格式数据到服务器端执行。比如下面的形式的数据（linux服务器上需要用unix2dos转化成dos文件）。 linux下默认的换行是\n,windows系统的换行符是\r\n，redis使用的是\r\n. echo -en

'*3\r\n$3\r\nSET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n' | redis-cli --pipe

9. -a

如果开启了requirepass，那么你如果希望调用或者自己编写一些外部脚本通过redis-cli进行操作或者监控redis，那么这个选项可以让你不用再手动输入auth。这个选项很普遍，但是往往被人忽视。

6.13 持久化与备份恢复

3.13.1 RDB相关操作

BGSAVE：后台子进程进行RDB持久化 SAVE：主进程进行RDB，生产环境千万别用，服务器将无法响应任何操作。 LASTSAVE： 返回上一次成功SAVE的Unix时间

动态关闭RDB：

```
redis-cli config set save ""
```

动态设置RDB：

```
redis-cli config set save "900 1"
```

永久关闭RDB：

```
sed -e '/save/ s/^#*/#/' -i /etc/redis/redis.conf
```

永久设置RDB：

```
在redis.conf中设置save选项
```

查看RDB是否打开：

```
redis-cli config get save
```

空的即是关闭，有数字的都是打开的。

3.13.2 AOF相关操作

```
BGREWRITEAOF
```

在后台执行一个 AOF文件重写操作

动态关闭AOF：

```
redis-cli config set appendonly no
```

动态打开AOF：

```
redis-cli config set appendonly yes
```

永久关闭AOF：

```
sed -e '/appendonly/ s/^#*/#/' -i /etc/redis/redis.conf  （默认是关闭的）
```

永久打开AOF：

```
将appendonly yes设置在redis.conf中
```

3.13.3 备份

对于RDB和AOF，都是直接拷贝文件即可，可以设定crontab进行定时备份： cp /var/lib/redis/dump.rdb /somewhere/safe/dump.$(date +%Y%m%d%H%M).rdb

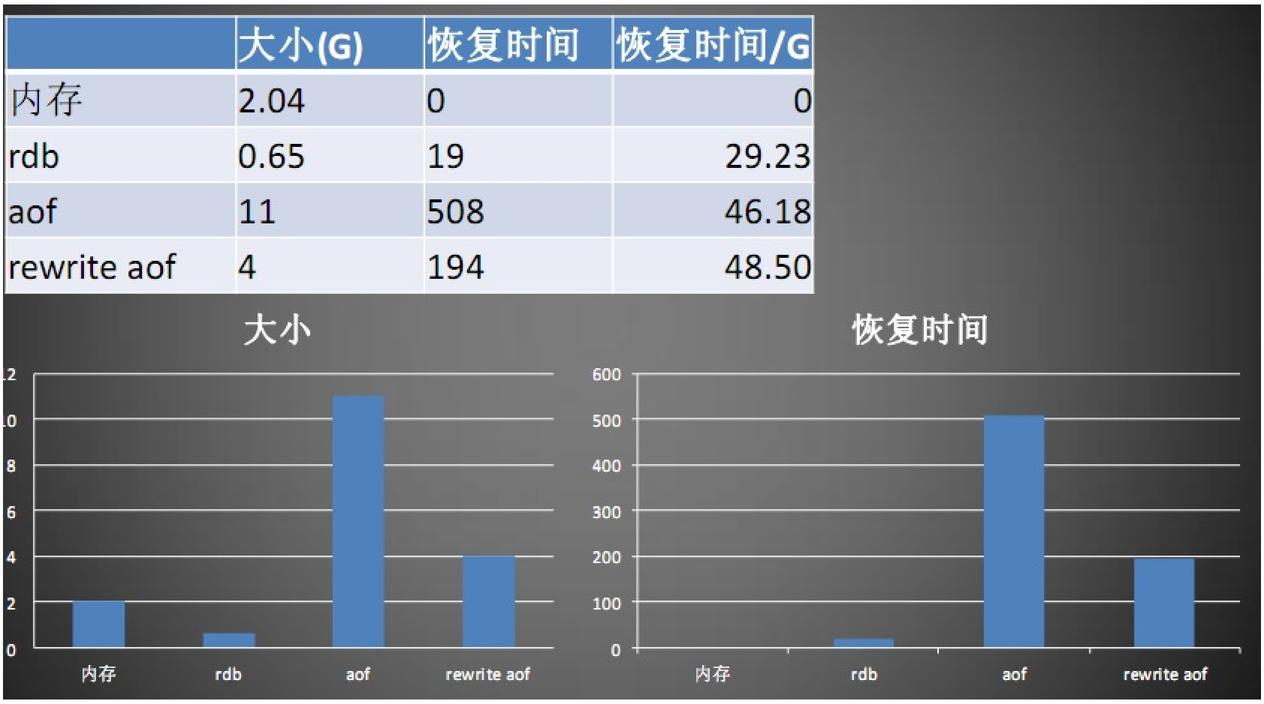
3.13.4 恢复

如果只使用了RDB，则首先将redis-server停掉，删除dump.rdb，最后将备份的dump.rdb文件拷贝回data目录并修改相关属主保证其属主和redis-server启动用户一致，然后启动redis-server。

如果是RDB+AOF的持久化，只需要将aof文件放入data目录，启动redis-server，查看是否恢复，如无法恢复则应该将aof关闭后重启，redis就会从rdb进行恢复了，随后调用命令BGREWRITEAOF进行AOF文件写入，在info的aof_rewrite_in_progress为0后一个新的aof文件就生成了，此时再将配置文件的aof打开，再次重启redis-server就可以恢复了。注意先不要将dump.rdb放入data目录，否则会因为aof文件万一不可用，则rdb也不会被恢复进内存，此时如果有新的请求进来后则原先的rdb文件被重写。

如果只配置了AOF，重启时加载AOF文件恢复数据。

恢复速度参见新浪的测试结果：



	大小(G)	恢复时间	恢复时间/G
内存	2.04	0	0
rdb	0.65	19	29.23
aof	11	508	46.18
rewrite aof	4	194	48.50

这个结果是可信的，在一台SSD、4个CPU的虚拟机上测试为28.3G/s.

检查修复AOF文件：

```
redis-check-aof data/appendonly.aof
```

7. 数据迁移

4.1 将key从当前数据库移动到指定数据库

```
move key db-index
```

返回1成功。0 如果key不存在，或者已经在指定数据库中

8. 数据迁移

8.1 一般处理流程

6.1.1 探测服务是否可用

```
127.0.0.1:6379> ping
```

返回PONG说明正常

6.1.2 探测服务延迟

redis-cli --latency 显示的单位是milliseconds，作为参考，千兆网一跳一般延迟为0.16ms左右

6.1.3 监控正在请求执行的命令

在cli下执行monitor，生产环境慎用。

6.1.4 查看统计信息

```
Mrds:6379> info
```

在cli下执行info。

```
info Replication
```

只看其中一部分。

```
config resetstat
```

重新统计。

```
# Server
redis_version:2.8.19 ###redis版本号
redis_git_sha1:00000000  ###git SHA1
redis_git_dirty:0   ###git dirty flag
redis_build_id:78796c63e58b72dc
redis_mode:standalone   ###redis运行模式
os:Linux 2.6.32-431.el6.x86_64 x86_64   ###os版本号
arch_bits:64  ###64位架构
multiplexing_api:epoll  ###调用epoll算法
gcc_version:4.4.7   ###gcc版本号
process_id:25899   ###服务器进程PID
run_id:eae356ac1098c13b68f2b00fd7e1c9f93b1c6a2c   ###Redis的随机标识符(用于sentinel和集群)
tcp_port:6379   ###Redis监听的端口号
uptime_in_seconds:6419 ###Redis运行时长(s为单位)
uptime_in_days:0  ###Redis运行时长(天为单位)
hz:10
lru_clock:10737922  ###以分钟为单位的自增时钟,用于LRU管理
config_file:/etc/redis/redis.conf   ###redis配置文件

# Clients
connected_clients:1   ###已连接客户端的数量（不包括通过从属服务器连接的客户端）这个参数也要一定关注，有飙升和明显下降时都会有问题。即使不操作
client_longest_output_list:0   ###当前连接的客户端中最长的输出列表
client_biggest_input_buf:0   ###当前连接的客户端中最大的。输出缓存
blocked_clients:0  ###正在等待阻塞命令（BLPOP、BRPOP、BRPOPLPUSH）的客户端的数量 需监控

# Memory
used_memory:2281560   ###由 Redis 分配器分配的内存总量，以字节（byte）为单位
used_memory_human:2.18M   ###以更友好的格式输出redis占用的内存
used_memory_rss:2699264   ###从操作系统的角度，返回 Redis 已分配的内存总量（俗称常驻集大小）。这个值和 top 、 ps 等命令的输出一致，包含了used_memory和内存碎片。
used_memory_peak:22141272  ### Redis 的内存消耗峰值（以字节为单位）
used_memory_peak_human:21.12M  ###以更友好的格式输出redis峰值内存占用
used_memory_lua:35840  ###LUA引擎所使用的内存大小
```

```
mem_fragmentation_ratio:1.18 ###  =used_memory_rss /used_memory 这两个参数都包含保存用户k-v数据的内存和redis内部不同数据结构需要占用的内存，并且RSS指的是包含操作系统给redis实例分配的内存，这里面还包含不连续分配所带来的开销。因此在理想情况下， used_memory_rss 的值应该只比 used_memory 稍微高一点儿。当 rss > used ，且两者的值相差较大时，表示存在（内部或外部的）内存碎片。内存碎片的比率可以通过 mem_fragmentation_ratio 的值看出。当 used > rss 时，表示 Redis 的部分内存被操作系统换出到交换空间了，在这种情况下，操作可能会产生明显的延迟。可以说这个值大于1.5或者小于1都是有问题的。当大于1.5的时候需要择机进行服务器重启。当小于1的时候需要对redis进行数据清理
mem_allocator:jemalloc-3.6.0

# Persistence
loading:0  ###记录服务器是否正在载入持久化文件，1为正在加载
rdb_changes_since_last_save:0   ###距离最近一次成功创建持久化文件之后，产生了多少次修改数据集的操作
rdb_bgsave_in_progress:0   ###记录了服务器是否正在创建 RDB 文件，1为正在进行
rdb_last_save_time:1420023749  ###最近一次成功创建 RDB 文件的 UNIX 时间戳
rdb_last_bgsave_status:ok   ###最近一次创建 RDB 文件的结果是成功还是失败,失败标识为err，这个时候写入redis 的操作可能会停止，因为默认stop-writes-on-bgsave-error是开启的，这个时候如果需要尽快恢复写操作，可以手工将这个选项设置为no。
rdb_last_bgsave_time_sec:0  ###最近一次创建 RDB 文件耗费的秒数
rdb_current_bgsave_time_sec:-1  ###如果服务器正在创建 RDB 文件，那么这个域记录的就是当前的创建操作已经耗费的秒数
aof_enabled:1   ###AOF 是否处于打开状态，1为启用
aof_rewrite_in_progress:0   ###服务器是否正在创建 AOF 文件
aof_rewrite_scheduled:0   ###RDB 文件创建完毕之后，是否需要执行预约的 AOF 重写操作（因为在RDB时aof的rewrite会被阻塞一直到RDB结束）
aof_last_rewrite_time_sec:-1  ###最近一次创建 AOF 文件耗费的时长
aof_current_rewrite_time_sec:-1  ###如果服务器正在创建 AOF 文件，那么这个域记录的就是当前的创建操作已经耗费的秒数
aof_last_bgrewrite_status:ok  ###最近一次创建 AOF 文件的结果是成功还是失败
aof_last_write_status:ok
aof_current_size:176265  ###AOF 文件目前的大小
aof_base_size:176265  ###服务器启动时或者 AOF 重写最近一次执行之后，AOF 文件的大小
aof_pending_rewrite:0  ###是否有 AOF 重写操作在等待 RDB 文件创建完毕之后执行
aof_buffer_length:0   ###AOF 缓冲区的大小
aof_rewrite_buffer_length:0  ###AOF 重写缓冲区的大小
aof_pending_bio_fsync:0  ###后台 I/O 队列里面，等待执行的 fsync 调用数量
aof_delayed_fsync:0###被延迟的 fsync 调用数量
loading_start_time:1441769386   loading启动时间戳
loading_total_bytes:1787767808   loading需要加载数据量
loading_loaded_bytes:1587418182  已经加载的数据量
loading_loaded_perc:88.79 加载百分比
loading_eta_seconds:7   剩余时间

# Stats
total_connections_received:8466  ###服务器已接受的连接请求数量，注意这是个累计值。
total_commands_processed:900668   ###服务器已执行的命令数量，这个数值需要持续监控，如果在一段时间内出现大范围波动说明系统要么出现大量请求，要么出现执行缓慢的操作。
instantaneous_ops_per_sec:1   ###服务器每秒钟执行的命令数量
total_net_input_bytes:82724170
total_net_output_bytes:39509080
instantaneous_input_kbps:0.07
instantaneous_output_kbps:0.02
rejected_connections:0  ###因为最大客户端数量限制而被拒绝的连接请求数量
```

```
sync_full:2
sync_partial_ok:0
sync_partial_err:0
expired_keys:0   ###因为过期而被自动删除的数据库键数量
evicted_keys:0   ###因为最大内存容量限制而被驱逐（evict）的键数量。这个数值如果不是0则说明maxmemo
ry被触发，并且    evicted_keys一直大于0，则系统的latency增加，此时可以临时提高最大内存，但这只是临
时措施，需要从应用着手分析。
keyspace_hits:0  ###查找数据库键成功的次数。可以计算命中率
keyspace_misses:500000   ###查找数据库键失败的次数。
pubsub_channels:0  ###目前被订阅的频道数量
pubsub_patterns:0  ###目前被订阅的模式数量
latest_fork_usec:402  ###最近一次 fork() 操作耗费的毫秒数

# Replication
role:master   ###如果当前服务器没有在复制任何其他服务器，那么这个域的值就是 master ；否则的话，这
个域的值就是 slave 。注意，在创建复制链的时候，一个从服务器也可能是另一个服务器的主服务器
connected_slaves:2   ###2个slaves
slave0:ip=192.168.65.130,port=6379,state=online,offset=1639,lag=1
slave1:ip=192.168.65.129,port=6379,state=online,offset=1639,lag=0
master_repl_offset:1639
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:1638

# CPU
used_cpu_sys:41.87  ###Redis 服务器耗费的系统 CPU
used_cpu_user:17.82  ###Redis 服务器耗费的用户 CPU
used_cpu_sys_children:0.01  ###后台进程耗费的系统 CPU
used_cpu_user_children:0.01  ###后台进程耗费的用户 CPU

# Keyspace
db0:keys=3101,expires=0,avg_ttl=0   ###keyspace 部分记录了数据库相关的统计信息，比如数据库的键
数量、数据库过期键数量等。对于每个数据库，这个部分都会添加一行此信息
```

6.1.5 获取慢查询

```
SLOWLOG GET 10
```

结果为查询ID、发生时间、运行时长和原命令 默认10毫秒，默认只保留最后的128条。单线程的模型下，一个请求占掉10毫秒是件大事情，注意设置和显示的单位为微秒，注意这个时间是不包含网络延迟的。

```
slowlog get
```

获取慢查询日志

```
slowlog len
```

获取慢查询日志条数

```
slowlog reset
```

清空慢查询

6.1.6 查看客户端

```
client list
```

列出所有连接

```
client kill
```

杀死某个连接， 例如`CLIENT KILL 127.0.0.1:43501`

```
client getname #
```

获取连接的名称 默认nil

```
client setname "名称"
```

设置连接名称,便于调试

6.1.7 查看日志

日志位置在/redis/log下，redis.log为redis主日志，sentinel.log为sentinel监控日志。

6.2 并发延迟检查

top看到单个CPU 100%时，就是垂直扩展的时候了。如果需要让CPU使用率最大化，可以配置Redis实例数对应CPU数, Redis实例数对应端口数(8核Cpu, 8个实例, 8个端口), 以提高并发。单机测试时, 单条数据在200字节, 测试的结果为8~9万tps。（未实测）。 另外，对于命令的复杂度一定要关注。

6.2.1 检查CPU情况

```
mpstat -P ALL 1
```

6.2.2 检查网络情况

可以在系统不繁忙或者临时下线前检测客户端和server或者proxy 的带宽：

1)使用 iperf -s 命令将 Iperf 启动为 server 模式:

```
iperf –s
——————————————————————
Server listening on TCP port 5001
TCP window size: 8.00 KByte (default)
——————————————————————
```

2)启动客户端，向IP为10.230.48.65的主机发出TCP测试，并每2秒返回一次测试结果，以Mbytes/sec为单位显示测试结果：

```
iperf -c 10.230.48.65 -f M -i 2
```

6.2.3 检查系统情况

着重检查探测服务延迟、 监控正在请求执行的命令、获取慢查询

6.2.4 检查连接数

查看info里面的total_connections_received，如果该值不断升高，则需要修改应用，采用连接池方式进行，因为频繁关闭再创建连接redis的开销很大。

6.2.5 检查持久化

RDB的时间： `latest_fork_usec:936` 上次导出rdb快照,持久化花费，微秒。 检查是否有人使用了SAVE。

6.2.6 检查命令执行情况

```
INFO commandstats
```

查看命令执行了多少次，执行命令所耗费的毫秒数(每个命令的总时间和平均时间)

8.3 内存检查

8.3.1 系统内存查看

script/下的memstat.sh或者ps_mem.py都可以查看系统的内存情况，两个工具都需要root权限。

8.3.2 系统swap内存查看

```
#!/bin/bash
# Get current swap usage for all running processes
# Erik Ljungstrom 27/05/2011
# Modified by Mikko Rantalainen 2012-08-09
# Pipe the output to "sort -nk3" to get sorted output
# Modified by Marc Methot 2014-09-18
# removed the need for sudo

SUM=0
OVERALL=0
for DIR in `find /proc/ -maxdepth 1 -type d -regex "^/proc/[0-9]+"`
do
    PID=`echo $DIR | cut -d / -f 3`
    PROGNAME=`ps -p $PID -o comm --no-headers`
for SWAP in `grep VmSwap $DIR/status 2>/dev/null | awk '{ print $2 }'`
do
    let SUM=$SUM+$SWAP
done
if (( $SUM > 0 )); then
    echo "PID=$PID swapped $SUM KB ($PROGNAME)"
fi
let OVERALL=$OVERALL+$SUM
SUM=0
done
echo "Overall swap used: $OVERALL KB"
```

8.3.3 info查看内存

`used_memory:859192` 数据结构的空间 `used_memory_rss:7634944` 实占空间 `mem_fragmentation_ratio:8.89` 前2者的比例,1.N为佳,如果此值过大,说明redis的内存的碎片化严重,可以导出再导入一次.

8.3.4 dump.rdb文件成生内存报告（rdb-tool）

```
# rdb -c memory ./dump.rdb > redis_memory_report.csv
# sort -t, -k4nr redis_memory_report.csv
```

8.3.5 query在线分析

```
redis-cli MONITOR | head -n 5000 | ./redis-faina.py
```

8.3.6 内存抽样分析

```
/redis/script/redis-sampler.rb 127.0.0.1 6379 0 10000
/redis/script/redis-audit.rb  127.0.0.1 6379 0 10000
```

8.3.7 统计生产上比较大的key

```
./redis-cli --bigkeys
```

对redis中的key进行采样，寻找较大的keys。是用的是scan方式，不用担心会阻塞redis很长时间不能处理其他的请求。执行的结果可以用于分析redis的内存的只用状态，每种类型key的平均大小。

8.3.8 查看key内部结构和编码等信息

```
debug object
```

查看一个key内部信息，比如refcount、encoding、serializedlength等，结果如下 Value at:0x7f21b9479850 refcount:1 encoding:raw serializedlength:6 lru:8462202 lru_seconds_idle:215

8.3.9 Rss增加，内存碎片增加

此时可以选择时间进行redis服务器的重新启动，并且注意在rss突然降低观察是否swap被使用，以确定并非是因为swap而导致的rss降低。

一个典型的例子是：http://grokbase.com/t/gg/redis-db/14ag5n9qhv/redis-memory-fragmentation-ratio-reached-5000

9. 测试方法

7.1 模拟oom

```
redis-cli debug oom
```

redis直接退出。

7.2 模拟宕机

```
redis-cli debug segfault
```

7.3 模拟hang

```
redis-cli -p 6379 DEBUG sleep 30
```

7.4 快速产生测试数据

```
debug populate
```

测试利器，快速产生大量的key

```
127.0.0.1:6379> debug populate 10000
OK
127.0.0.1:6379> dbsize
(integer) 10000
```

7.5 模拟RDB load情形

```
debug reload
```

save当前的rdb文件，并清空当前数据库，重新加载rdb，加载与启动时加载类似，加载过程中只能服务部分只读请求（比如info、ping等）： rdbSave(); emptyDb(); rdbLoad();

7.6 模拟AOF加载情形

```
debug loadaof
```

清空当前数据库，重新从aof文件里加载数据库 emptyDb(); loadAppendOnlyFile();

9. Redis安全问题

Redis是一个弱安全的组件，只有一个简单的明文密码，因此在保护上需要对其他多方面的措施，另外，很多所谓安全问题不是redis本身造成的，而是误用的结果。

9.1 Shell提权问题

问题报告：http://drops.wooyun.org/papers/3062

问题分析：Redis 安全模型的观念是: “请不要将Redis暴露在公开网络中, 因为让不受信任的客户接触到Redis是非常危险的” 。The Redis security model is: “it’s totally insecure to let untrusted clients access the system, please protect it from the outside world yourself”. 因此最近爆出的问题也非redis本身产品问题，属于不当配置。

问题规避：

1. 使用redis单独用户和组进行安全部署，并且在OS层面禁止此用户ssh登陆，这就从根本上防止了root用户启停redis带来的风险。
2. 修改默认端口，降低网络简单扫描危害。
3. 修改绑定地址，如果是本地访问要求绑定本地回环。
4. 要求设置密码，并对配置文件访问权限进行控制，因为密码在其中是明文。
5. HA环境下主从均要求设置密码。 另外，我们建议在网络防火墙层面进行保护，杜绝任何部署在外网直接可以访问的redis的出现。

10. 简述

10.1 概念

在本文档中，高可用主要指的是解决尽可能在不丢失数据的前提下不间断服务问题，由于redis是异步复制，因此不保证数据完全不丢失，在这个场景下并不实现动态横向扩容，只能进行纵向扩容，你只要加内存，启动redis，设置maxmemory即可。而分片（Sharding）主要指的是解决在线动态横向扩容缩容问题，当然为了稳定也进行高可用部署配置，即包含成对的主从关系。

10.2 高可用主要场景和对应思路

适用于redis非重度用户，内存占用不大，总体内存大小的增长趋势可预估，有一定停机时间的系统——纵向扩容即可满足，可以对全库进行主从复制即满足需求而不需要做分片，一般针对单个小型项目的cache 等场景。一般采用一主多从的sentinel方案进行部署。

10.3 分片主要场景和对应思路

分片是为了应对业务增长带来的数据增长，需要对动态横向扩容有一定要求时采用。对于一般的分片采用一致性哈希，它极大的优化机器增删时带来的哈希目标漂移问题。同时对于Hash目标漂移时产生的严重的数据倾斜，可以利用虚拟节点来优化。基本上，物理节点有了一定规模后，只要不是同时挂多个节点，或者同时扩容多个节点，数据分片不会有太大的扰动。穿透过Cache的请求后端存储可以抗住即可。

稍微复杂的方案是可以使用“预分片（Pre-Sharding)”的方案，也称为按“桶”进行数据划分，即分配一个相当大的集合，对Key哈希的结果落在这个集合中，集合的每个元素又与具体的物理节点存在多对一的路由映射关系，这张路由表由一个配置中心进行维护。其实，一致性哈希中的虚拟节点，实际上也可以归类到Pre-Sharding方案中。换句话说，只要是key经过两次哈希，第一次Hash到虚拟节点，第二次Hash到物理节点，都可以算作Pre-Sharding。只不过区别在于，一致性哈希的第二次Hash其路由表是按照算法固定的，而分桶的第二次Hash其路由表是第三方可配的。

10.4 适用场景对比列表

---	动态扩容能力	系统复杂度	开发复杂度	运维复杂度
主从复制+Sentinel	No	简单	简单	简单
Twemproxy	No	简单	简单	稍微复杂
3.0 Cluster	Yes	简单	简单	复杂
Codis	Yes	复杂	简单	复杂
应用层面 presharding	Yes	复杂	复杂	视开发的水平而定

11. 高可用和集群架构与实践

11.1 主从复制-sentinel架构

11.1.1 高可用原理

12.1.1.1 发现原理

Sentinel发现分为发现从服务器和发现其他sentinel服务两类：

- Sentinel实例可以通过询问主实例来获得所有从实例的信息
- Sentinel进程可以通过发布与订阅来自动发现正在监视相同主实例的其他Sentinel，每个Sentinel 都订阅了被它监视的所有主服务器和从服务器的 __sentinel__:hello 频道， 查找之前未出现过的 sentinel进程。 当一个 Sentinel 发现一个新的 Sentinel 时，它会将新的 Sentinel 添加到一个列表中，这个列表保存了 Sentinel 已知的，监视同一个主服务器的所有其他Sentinel。

注：文中的横杠要替换为下滑线，markdown不太能显示出来...

11.1.1.2 基本切换原理

在切换中，配置文件是会被动态修改的，例如当发生主备切换时候，配置文件中的master会被修改为另外一个slave。这样，之后sentinel如果重启时，就可以根据这个配置来恢复其之前所监控的redis集群的状态。 在sentinel切换过程中有三大步骤：

1. 判断是否下线（老主是否真的咽气驾崩） 每个sentinel在监控的时候，每秒对主进行一次ping命令，如果多次ping的响应时间超过了配置文件中的down-after-milliseconds，那么这个哨兵就会认为被监控的实例是SDown状态（Subjectively Down，主观down，SDOWN）。 这个时候此sentinel会判断此master是否真的挂了——即可以设置成ODOWN（Objectively Down，客观down，ODOWN）。设置成ODOWN的条件是除了当前sentinel认为此master SDOWN，还必须有其他sentinel认为此master SDOWN，当认为SDOWN的sentinel的个数等于或超过配置文件中monitor master最后的那个参数quorum后，就sentinel就会认为此master是ODOWN。 被标记为ODOWN的另一个效应是：在一般情况下，每个 Sentinel 进程会以每 10 秒一次的频率向它已知的所有主实例和从实例发送 INFO 命令。 当一个主实例被 Sentinel实例标记为客观下线时， Sentinel 向ODOWN Master的所有从实例发送 INFO 命令的频率会从 10 秒一次改为每秒一次。
2. 进行投票选举主持切换的sentinel（选举一个长老，由它来钦点新帝王） 当master被认为是ODOWN的时候，可能需要进行failover，但是并不是odown了就可以执行failover，因为可能有多个sentinel都认为master是odown了，这时候就需要选举一个sentiel来执行failover。也就是说切换之前要先选举一个sentinel来主持切换，条件是必须有>=max(quorum, num(sentinels)/2 +1)的sentinel都同意某一个sentinel主持failover，那么这个sentinel就可以主持failover，超过半数这个条件就能限制住此时刻只有一个sentinel来操作。 这个也是通过is-master-down-by-addr消息进行更新每个sentinel的选举的leader。每个sentinel都有一个epoch，这个东西相当于一个时间戳，是递增的值，如果集群正常的话，所有的sentinel的这个值都是一样的。当master出现异常后，每个sentinel后自增这个值，如果一直没有选举出来leader的话，这个值会跟随这time event的轮询，每次加一，在设定的故障迁移超时时间的两倍之后， 重新尝试当选。同时is-master-down-by-addr会把这个值发送到其他的sentinel，其他的sentinel收到这个消息后，会判断自己的epoch和消息中的epoch，如果自己的epoch小于消息中的epoch，那么其他的sentinel就会选举传递消息的这个sentinel。最终会大部分sentinel都同意一个较大的epoch的sentinel主持failover。

3. 进行切换，并其他实例同步新Master（新帝王登基，其余藩王宣誓效忠新帝王） 这个主持切换的sentinel选出一个从redis实例，并将它升级为Master。首先是要下面的条件按照如下条件筛选备选node： 1) slave节点状态处于S_DOWN,O_DOWN,DISCONNECTED的除外 2) 最近一次ping应答时间不超过5倍ping的间隔（假如ping的间隔为1秒，则最近一次应答延迟不应超过5秒，redis sentinel默认为1秒） 3) info_refresh应答不超过3倍info_refresh的间隔（原理同2,redis sentinel默认为10秒） 4) slave节点与master节点失

去联系的时间不能超过（ (now - master->s_down_since_time) + (master->down_after_period * 10)）。总体意思是说，slave节点与master同步太不及时的（比如新启动的节点），不应该参与被选举。 5) Slave priority不等于0（这个是在配置文件中指定，默认配置为100）。

然后再从备选node中，按照如下顺序选择新的master 1) 较低的slave_priority（这个是在配置文件中指定，默认配置为100） 2) 较大的replication offset（每个slave在与master同步后offset自动增加） 3) 较小的runid（每个redis实例，都会有一个runid,通常是一个40位的随机字符串,在redis启动时设置，重复概率非常小） 4) 如果以上条件都不足以区别出唯一的节点，则会看哪个slave节点处理之前master发送的command多，就选谁。 主持切换的sentinel向被选中的从redis实例发送 SLAVEOF NO ONE 命令，让它转变为Master。然后通过发布与订阅功能，将更新后的配置传播给所有其他 Sentinel，其他 Sentinel 对它们自己的配置进行config-rewrite。随后sentinel向已下线的Master的从服务器发送SLAVEOF命令，让它们去复制新Master。

注意：sentinel failover-timeout这个选项有四个含义，有必要在此翻译一下 1) 对于一个sentinel选出的同一个master进行再次的failover尝试所需要的时间——这个参数值的两倍。 2) 当sentinel发现一个slave错误的复制了一个错的主时sentinel会强迫其复制正确的主的时间。 3) 取消一个已经开始但是还没有引起任何配置改变的failover所需要的时间。 4) 等待所有slave被重新配置为新主的slave而所需要的最大时间。注意即使超过了这个时间sentinel也会最终配置slave去同步最新的master

11.1.2 环境搭建

11.1.2.1 部署架构

部署架构上采用三台机器，一个Master接受写请求，两个Slave进行数据同步，三台机器上都部署sentinel（一般为奇数个，因为需要绝大部分进行投票才能failover）。（官方示例）具体架构如下图：

```
       +----+
       | M1 |
       | S1 |
       +----+
          |
+----+    |    +----+
| R2 |----+----| R3 |
| S2 |         | S3 |
+----+         +----+

Configuration: quorum = 2
```

注意：如果有条件可以将sentinel多部署几个在客户端所在的应用服务器上，而不是与从节点部署在一起，这样避免整机宕机后sentinel和slave都减少而导致的切换选举sentinel无法超过半数。

11.1.2.2 网络规划

redis高可用环境不需要进行心跳线的配置，每个物理节点的网卡进行双网卡主备绑定生成bond0即可。

11.1.2.3 用户规划

用户名	用户所在组	用户目录	权限	备注
redis(10086)	redis (10086)	/redis	sudo（如需要浮动IP时赋予）	--

11.1.2.4 持久化规划

如果读多写少，可以在master上只开启aof，在低峰期定时进行bgsave，在slave上彻底关闭持久化。 如果读写差不多，可以在一个slave上开启rdb（这个slave只做持久化，不进行读操作），在其余主从都关闭持久化。 注意：从节点是不会从本地恢复而直接从master节点进行恢复的，因此在重启前如果有需要备份从节点，则需要把aof和rdb文件移走。

11.1.2.5 目录规划

目录	含义
/redis/bin	redis可执行文件
/redis/conf	redis 和supervisord的配置文件
/redis/run	redis和supervisord运行时的pid文件
/redis/log	redis和supervisord的日志
/redis/script	一些管理脚本和测试脚本
/redis/data	Redis持久化数据目录

11.1.2.6 部署步骤

解压下列压缩包至/tmp/redis目录，以符合上述目录结构：

部署相关组件： cd /tmp/redis/deploy ./deploy.sh

修改Master配置文件redis.conf，注释掉包含slaveof的语句。 修改Slave配置文件redis.conf，添加slaveof masterIP port，指定主从 修改三台机器的sentinel配置文件，指定主服务器的IP和端口： sentinel monitor mymaster masterIP port 2

然后使用supervisord重新启动。

11.1.2.7 配置文件

首先，一个sentinel可以配置多个master。一个master的配置如下：

```
port 26379
###定义目录存放
dir "/redis"
###监控mymaster(可自定义-但只能包括A-z 0-9和”._-”)，注意quorum只影响ODOWN的判断，但是不影响failover，发生failover的条件必须是半数sentinel认为老Master已经ODOWN。此参数建议设置为sentinel/2+1的数值，否则可能会产生脑裂。
sentinel monitor mymaster 192.168.145.131 6379 2
###mymaster多久不响应认为SDOWN，设置为3100也就是说3次ping失败后认为SDOWN
sentinel down-after-milliseconds mymaster 3100
###如果在该时间（ms）内未能完成failover操作，则认为该failover失败
sentinel failover-timeout mymaster 15000

###在执行故障转移时， 最多可以有多少个从Redis实例在同步新的主实例， 在从Redis实例较多的情况下这个数字越小，同步的时间越长，完成故障转移所需的时间就越长
sentinel parallel-syncs mymaster 1

###reconfig的时候执行的脚本（选配）
sentinel client-reconfig-script mymaster /redis/script/failover.sh

###出现任何sentinel在warning事件时候执行的脚本（选配）
sentinel notification-script mymaster  /redis/script/notify.sh

####日志位置
logfile "/redis/log/sentinel.log"
```

11.1.3 维护操作

11.1.3.1 完整启动

supervisord -c /redis/conf/redis-supervisord.conf 会自动拉起本机的redis和sentinel

11.1.3.2 启停redis

supervisorctl -c /redis/conf/redis-supervisord.conf start redis supervisorctl -c /redis/conf/redis-supervisord.conf stop redis supervisorctl -c /redis/conf/redis-supervisord.conf restart redis

11.1.3.3 手动启动

有两种方式： 第一种：redis-sentinel /path/to/sentinel.conf 第二种：redis-server /path/to/sentinel.conf --sentinel

11.1.3.4 启停sentinel

supervisorctl -c /redis/conf/redis-supervisord.conf start redis-sentinel supervisorctl -c /redis/conf/redis-supervisord.conf stop redis-sentinel supervisorctl -c /redis/conf/redis-supervisord.conf restart redis-sentinel

11.1.3.5 查看sentinel状态

redis-cli -p 26379 info

11.1.3.6 查看master地址和端口

sentinel get-master-addr-by-name myredis

11.1.3.7 查看master配置

redis-cli -p 26379 sentinel masters

11.1.3.8 重置该sentinel

sentinel reset myredis 重置操作清除该sentinel的所保存的所有状态信息，并进行一次重新的发现过程。

11.1.3.9 动态修改sentinel配置

SENTINEL SET command 例如：

```
SENTINEL SET objects-cache-master down-after-milliseconds 1000
```

11.1.3.10 主动切换

sentinel failover myredis 此操作会将新的配置发送到其他sentinel上。

11.1.3.11 判断主从是否完全一致

```
dbsize
```

查看key 的数目

```
debug digest
```

对整个数据库的数据，产生一个摘要，可用于验证两个redis数据库数据是否一致
127.0.0.1:6379> debug digest 7164ae8b6730c8bcade46532e5e4a8015d4cccfb
127.0.0.1:6379> debug digest 7164ae8b6730c8bcade46532e5e4a8015d4cccfb

11.1.3.12 接收所有事件信息

```
127.0.0.1:26379>  PSUBSCRIBE *
```

注意这是在sentinel上监控所有的频道信息，查看的是切换前后发生的消息。

还有一个__sentinel__:hello的频道，这个频道是在redis实例上的，用途是sentinel之间发现对方的，别无它用。在redis实例上可以通过monitor或者订阅此频道看到这个消息。

11.1.4 高可用和异常测试

11.1.4.1 测试环境介绍

```
Master：192.168.2.128 （A）:6379
Slave：192.168.2.129  （B）:6379
Slave：192.168.2.130  （B）:6379
Sentinel：三台机器的26379端口
```

sentinel的消息可以通过sentinel日志（/redis/log/sentinel.log）以及**sentinel**:hello订阅此频道进行查看。

11.1.4.2 手动切换测试

集群情况，2.128为主

```
# Replication
role:master
connected_slaves:2
min_slaves_good_slaves:2
slave0:ip=192.168.2.130,port=6379,state=online,offset=14617637,lag=0
slave1:ip=192.168.2.129,port=6379,state=online,offset=14617637,lag=0
```

发起主动切换：

```
127.0.0.1:26379> sentinel failover mymaster
OK
```

查看sentinel日志：

```
    [1158] 19 Jun 08:14:38.504 # Executing user requested FAILOVER of 'mymaster'
    [1158] 19 Jun 08:14:38.507 # +new-epoch 29
    [1158] 19 Jun 08:14:38.507 # +try-failover master mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:38.581 # +vote-for-leader 7d60ccf8a9f9f81e5292a0dbde2c54c76a2b
d265 29
    [1158] 19 Jun 08:14:38.581 # +elected-leader master mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:38.581 # +failover-state-select-slave master mymaster 192.168.
2.129 6379
    [1158] 19 Jun 08:14:38.655 # +selected-slave slave 192.168.2.128:6379 192.168.2.12
8 6379 @ mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:38.655 * +failover-state-send-slaveof-noone slave 192.168.2.12
8:6379 192.168.2.128 6379 @ mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:38.714 * +failover-state-wait-promotion slave 192.168.2.128:63
79 192.168.2.128 6379 @ mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:39.642 # +promoted-slave slave 192.168.2.128:6379 192.168.2.12
8 6379 @ mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:39.642 # +failover-state-reconf-slaves master mymaster 192.168
.2.129 6379
    [1158] 19 Jun 08:14:39.705 * +slave-reconf-sent slave 192.168.2.130:6379 192.168.2
.130 6379 @ mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:40.645 * +slave-reconf-inprog slave 192.168.2.130:6379 192.168
.2.130 6379 @ mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:40.645 * +slave-reconf-done slave 192.168.2.130:6379 192.168.2
.130 6379 @ mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:40.735 # +failover-end master mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:14:40.735 # +switch-master mymaster 192.168.2.129 6379 192.168.2.
128 6379
    [1158] 19 Jun 08:14:40.736 * +slave slave 192.168.2.130:6379 192.168.2.130 6379 @
mymaster 192.168.2.128 6379
    [1158] 19 Jun 08:14:40.743 * +slave slave 192.168.2.129:6379 192.168.2.129 6379 @
mymaster 192.168.2.128 6379
    [1158] 19 Jun 08:27:56.524 # +new-epoch 30
    [1158] 19 Jun 08:27:57.519 # +config-update-from sentinel 192.168.2.128:26379 192.
168.2.128 26379 @ mymaster 192.168.2.128 6379
    [1158] 19 Jun 08:27:57.519 # +switch-master mymaster 192.168.2.128 6379 192.168.2.
129 6379
    [1158] 19 Jun 08:27:57.519 * +slave slave 192.168.2.130:6379 192.168.2.130 6379 @
mymaster 192.168.2.129 6379
    [1158] 19 Jun 08:27:57.524 * +slave slave 192.168.2.128:6379 192.168.2.128 6379 @
mymaster 192.168.2.129 6379
```

在2.129上看，集群已经切换过来：

```
# Replication
role:master
connected_slaves:2
slave0:ip=192.168.2.130,port=6379,state=online,offset=14774926,lag=1
slave1:ip=192.168.2.128,port=6379,state=online,offset=14774926,lag=1
```

11.1.4.3 主实例宕测试

接上，此时master为2.129，找出redis实例的pid，然后kill：

```
[root@hadoop2 log]# ps -ef |grep redis-server
root      11349   1157  1 Jun18 ?        00:15:45 /usr/bin/redis-server 0.0.0.0:6379
root      14969  10433  0 08:33 pts/1    00:00:00 grep --color=auto redis-server
[root@hadoop2 log]# kill 11349
```

此时查看sentinel日志：

```
[1158] 19 Jun 08:33:57.953 # +sdown master mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:58.025 # +odown master mymaster 192.168.2.129 6379 #quorum 3/2
[1158] 19 Jun 08:33:58.025 # +new-epoch 31
[1158] 19 Jun 08:33:58.025 # +try-failover master mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:58.028 # +vote-for-leader 7d60ccf8a9f9f81e5292a0dbde2c54c76a2bd265 31
[1158] 19 Jun 08:33:58.036 # 192.168.2.130:26379 voted for 7d60ccf8a9f9f81e5292a0dbde2c54c76a2bd265 31
[1158] 19 Jun 08:33:58.037 # 192.168.2.128:26379 voted for 7d60ccf8a9f9f81e5292a0dbde2c54c76a2bd265 31
[1158] 19 Jun 08:33:58.105 # +elected-leader master mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:58.105 # +failover-state-select-slave master mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:58.183 # +selected-slave slave 192.168.2.128:6379 192.168.2.128 6379 @ mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:58.183 * +failover-state-send-slaveof-noone slave 192.168.2.128:6379 192.168.2.128 6379 @ mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:58.267 * +failover-state-wait-promotion slave 192.168.2.128:6379 192.168.2.128 6379 @ mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:59.039 # +promoted-slave slave 192.168.2.128:6379 192.168.2.128 6379 @ mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:59.040 # +failover-state-reconf-slaves master mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:59.104 * +slave-reconf-sent slave 192.168.2.130:6379 192.168.2.130 6379 @ mymaster 192.168.2.129 6379
[1158] 19 Jun 08:33:59.245 # -odown master mymaster 192.168.2.129 6379
[1158] 19 Jun 08:34:00.082 * +slave-reconf-inprog slave 192.168.2.130:6379 192.168.2.130 6379 @ mymaster 192.168.2.129 6379
[1158] 19 Jun 08:34:00.082 * +slave-reconf-done slave 192.168.2.130:6379 192.168.2.130 6379 @ mymaster 192.168.2.129 6379
[1158] 19 Jun 08:34:00.193 # +failover-end master mymaster 192.168.2.129 6379
[1158] 19 Jun 08:34:00.193 # +switch-master mymaster 192.168.2.129 6379 192.168.2.128 6379
[1158] 19 Jun 08:34:00.194 * +slave slave 192.168.2.130:6379 192.168.2.130 6379 @ mymaster 192.168.2.128 6379
[1158] 19 Jun 08:34:00.200 * +slave slave 192.168.2.129:6379 192.168.2.129 6379 @ mymaster 192.168.2.128 6379
[1158] 19 Jun 08:34:03.319 # +sdown slave 192.168.2.129:6379 192.168.2.129 6379 @ mymaster 192.168.2.128 6379
```

从日志中可以看出已经切换到2.128，此时在2.128上看集群状态：

```
# Replication
role:master
connected_slaves:1
min_slaves_good_slaves:1
slave0:ip=192.168.2.130,port=6379,state=online,offset=30841,lag=0
```

目前2.128为主，2.130为从，2.129上的redis宕掉。现在重启2.129上的redis实例，启动后该节点会从原先的主变为从，并对2.128进行同步，最后达到同步状态：

```
# Replication
role:slave
master_host:192.168.2.128
master_port:6379
master_link_status:up
```

查看redis.conf和redis-sentinel.conf，发现都被改写。

11.1.4.4 单从实例宕测试

接上，2.129为从，此时杀掉该进程，redis.log日志记录如下：

```
[14984 | signal handler] (1434674492) Received SIGTERM scheduling shutdown...
[14984] 19 Jun 08:41:32.545 # User requested shutdown...
[14984] 19 Jun 08:41:32.545 * Calling fsync() on the AOF file.
[14984] 19 Jun 08:41:32.545 * Saving the final RDB snapshot before exiting.
[14984] 19 Jun 08:41:32.580 * DB saved on disk
[14984] 19 Jun 08:41:32.580 # Redis is now ready to exit, bye bye...
```

此时集群正常提供对外服务，并不影响。

11.1.4.5 双从实例宕测试

接上，此时Master为2.128，还有一个活着的从2.130，集群状态如下：

```
# Replication
role:master
connected_slaves:1
min_slaves_good_slaves:1
slave0:ip=192.168.2.130,port=6379,state=online,offset=154174,lag=0
```

此时，杀掉2.130的redis实例后，集群状态如下：

```
# Replication
role:master
connected_slaves:0
```

此时由于配置了最小slave个数为1，已经不满足，因此集群变为只读状态：

```
127.0.0.1:6379> get 38875
"tattkaqovx"
127.0.0.1:6379> set 38875 abcdad
(error) NOREPLICAS Not enough good slaves to write.
```

11.1.4.6 单sentinel宕测试

恢复集群状态，2.128为主，2.129、2.130为从。

```
# Replication
role:master
connected_slaves:2
min_slaves_good_slaves:2
slave0:ip=192.168.2.129,port=6379,state=online,offset=195839,lag=1
slave1:ip=192.168.2.130,port=6379,state=online,offset=195839,lag=1
```

此时，从2.128上看sentinel状态：

```
# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
master0:name=mymaster,status=ok,address=192.168.2.128:6379,slaves=2,sentinels=3
```

由于sentinel都是对等的，在此选择对2.128上的sentinel进行进程宕测试：

```
[root@hadoop1 ~]# supervisorctl stop redis-sentinel
redis-sentinel: stopped
[root@hadoop1 ~]# ps -ef |grep sentinel
root        1419   1375  0 00:06 pts/0    00:00:00 grep --color=auto sentinel
```

此时，本节点sentinel日志为：

```
[1248] 20 Jun 00:06:27.081 # User requested shutdown...
[1248] 20 Jun 00:06:27.081 # Sentinel is now ready to exit, bye bye...
```

其他节点sentinel日志为：

```
[1120] 20 Jun 00:06:29.839 # +sdown sentinel 192.168.2.128:26379 192.168.2.128 26379 @ mymaster 192.168.2.128 6
379
```

表示2.128上的sentnel已经宕。 此时集群读写正常，在一个sentinel宕机的基础上宕master后切换正常。

11.1.4.7 双sentinel宕测试

恢复集群状态，2.128为主，2.129、2.130为从。此时，将2.128的sentinel和2.129的sentinel都宕掉。此时主从集群读写均正常。 在双方sentinel宕机时，杀掉master，主从集群切换失效，原因是因为设置sentinel 的quorum为2，最少有两个sentinel活集群才正常切换。

11.1.4.8 master所在主机整体宕测试

恢复集群状态，2.128为主，2.129、2.130为从。此时，对2.128进行宕机测试，直接关闭电源。 主从切换至2.130,从2.129指向新的主：

```
# Replication
role:slave
master_host:192.168.2.129
master_port:6379
master_link_status:up
master_last_io_seconds_ago:0
master_sync_in_progress:0
slave_repl_offset:17771
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

sentinel日志为：

```
[1430] 20 Jun 00:38:50.864 # +sdown master mymaster 192.168.2.128 6379
[1430] 20 Jun 00:38:50.865 # +sdown sentinel 192.168.2.128:26379 192.168.2.128 26379 @ mymaster 192.168.2.128 6
379
[1430] 20 Jun 00:38:51.074 # +new-epoch 34
[1430] 20 Jun 00:38:51.077 # +vote-for-leader 740df7bb5ec393941b9591999e195dffc231d077 34
[1430] 20 Jun 00:38:52.003 # +odown master mymaster 192.168.2.128 6379 #quorum 2/2
[1430] 20 Jun 00:38:52.003 # Next failover delay: I will not start a failover before Sat Jun 20 00:39:21 2015
[1430] 20 Jun 00:38:52.148 # +config-update-from sentinel 192.168.2.129:26379 192.168.2.129 26379 @ mymaster 19
2.168.2.128 6379
[1430] 20 Jun 00:38:52.148 # +switch-master mymaster 192.168.2.128 6379 192.168.2.129 6379
[1430] 20 Jun 00:38:52.152 * +slave slave 192.168.2.130:6379 192.168.2.130 6379 @ mymaster 192.168.2.129 6379
[1430] 20 Jun 00:38:52.155 * +slave slave 192.168.2.128:6379 192.168.2.128 6379 @ mymaster 192.168.2.129 6379
[1430] 20 Jun 00:38:55.294 # +sdown slave 192.168.2.128:6379 192.168.2.128 6379 @ mymaster 192.168.2.129 6379
```

11.1.4.9 slave所在主机整体宕测试

恢复集群状态，2.128为主，2.129、2.130为从。此时直接关闭2.129，这时相当于一个redis slave进程和一个sentinel进程宕。主不受影响，并且感知到一个从已经宕机。

```
# Replication
role:master
connected_slaves:1
min_slaves_good_slaves:1
slave0:ip=192.168.2.130,port=6379,state=online,offset=68427,lag=0
master_repl_offset:68427
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:68426
```

sentinel日志记录了此事件。

```
[1249] 23 Jun 22:14:07.993 # +sdown sentinel 192.168.2.129:26379 192.168.2.129 26379 @ mymaster 192.168.2.128 6
379
[1249] 23 Jun 22:14:08.049 # +sdown slave 192.168.2.129:6379 192.168.2.129 6379 @ mymaster 192.168.2.128 6379
```

11.1.4.10 脑裂测试

恢复集群状态，2.128为主，2.129、2.130为从。首先进行一个从网络分离的测试：

```
                +-------------+
                | Sentinel 1  | <--- Client A
                | Redis 1 (M) |
                +-------------+
                        |
                        |
+-------------+         |                   +------------+
| Sentinel 2  |-----+-- / partition / ----| Sentinel 3 | <--- Client B
| Redis 2 (S) |                             | Redis 3 (M)|
+-------------+                             +------------+
```

此时集群状态为（从master看）：

```
# Replication
role:master
connected_slaves:2
min_slaves_good_slaves:2
slave0:ip=192.168.2.130,port=6379,state=online,offset=28595,lag=0
slave1:ip=192.168.2.129,port=6379,state=online,offset=28595,lag=0
master_repl_offset:28595
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:24176
repl_backlog_histlen:4420
```

此时切断2.130这个链路，2.128和2.129分别为主从形成一个集群，2.130会失败，因为没有足够的sentinel进行投票完成failover。剩余集群如下：

```
# Replication
role:master
connected_slaves:1
min_slaves_good_slaves:1
slave0:ip=192.168.2.129,port=6379,state=online,offset=891097,lag=0
master_repl_offset:904641
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:24176
repl_backlog_histlen:880466
```

第三台机器则为slave失败状态：

```
# Replication
role:slave
master_host:192.168.2.128
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
slave_repl_offset:341651
master_link_down_since_seconds:85
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

此时由于没有发生切换，因此对应用没有影响。

另一种情况，如果将主机网络断开，剩余两个从成为一个新的集群，其中一个从（2.129）成为主：

```
# Replication
role:master
connected_slaves:0
master_repl_offset:1033006
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1013009
repl_backlog_histlen:19998
```

原来的主机则为没有slave的主：

```
127.0.0.1:6379> set key1 abcd
(error) NOREPLICAS Not enough good slaves to write.
```

此时由于没有可用的slave，旧主无法写入（实际上由于网络断开也根本无法访问，因此从网络和数据库本身都不具有可写性）：

```
127.0.0.1:6379>  set huangpengcheng:email gnuhpc@gmail.com
OK
```

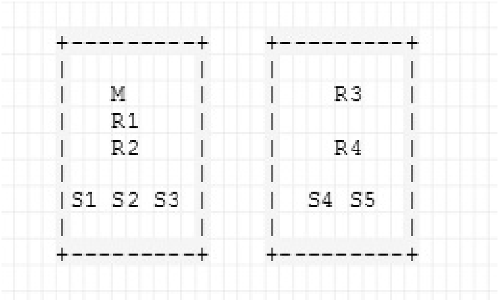
新主从可以接受读写请求：

```
127.0.0.1:6379> get huangpengcheng:email
"gnuhpc@gmail.com"
```

此时如果旧主的网络恢复，由于它的epoch比较旧，因此会成为从，将部分同步（psync）网络宕期间产生的新数据。

从上述两种情况测试，此架构不会导致双主对外服务，也不会因为网络恢复而数据混乱。

脑裂的场景还可以进行的一个测试时多个sentinel，例如下列架构（为了便于测试在两台机器上开多端口模拟多台机器）：



这个场景配置Quorum=3. 此时切断两台机器的通信网络（模拟两个机房之间通信中断），左边的机器（模拟主机房）集群不会受到影响，右边的机器（模拟灾备机房）由于不够大多数因此不会产生新的Master。

11.1.4.11 quorum测试

在一个如下的四节点环境中，

```
+--------+    +---------+    +---------+    +---------+
|        |    |         |    |         |    |         |
|   M1   |    |   R1    |    |   R2    |    |   R3    |
|        |    |         |    |         |    |         |
|   S1   |    |   S2    |    |   S3    |    |   S4    |
+--------+    +---------+    +---------+    +---------+
```

如果sentinel monitor的quorum设置为3，则宕机一台后再宕机，此时还剩余两台，存在两个sentinel，两个slave。由于quorum为3，而必须有>=max(quorum, num(sentinels)/2 +1) = max(3,2) = 3个sentinel都同意其中某一个sentinel主持failover，因此此时无sentinel可主持切换，因此测试表明，没有新的master被选出来，此时只能手动通过slaveof命令设置主从，并且手动切换（redis、sentinel和都应用不用重启）：

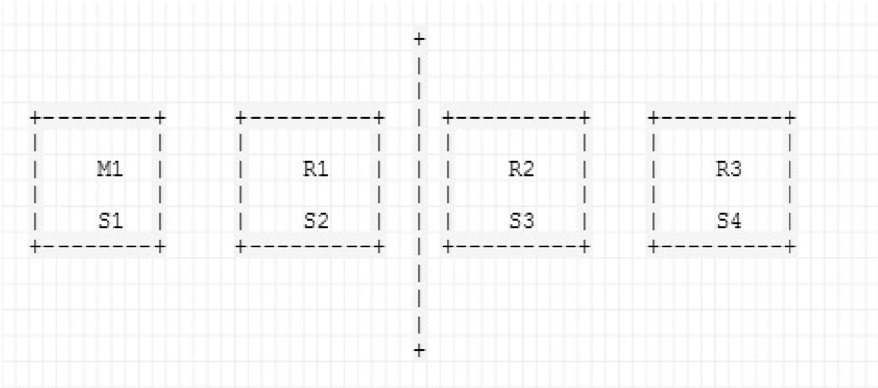
```
首先修改redis：
任意选取剩余的其中一个节点进行：slaveof no one
其他节点：slaveof 192.168.145.135 6379

找一个从节点上的sentinel，进入sentinel：
redis-cli -p 26379
进行主动切换：
sentinel failover mymaster
然后再在两个sentinel上重新发现集群：
sentinel reset mymaster

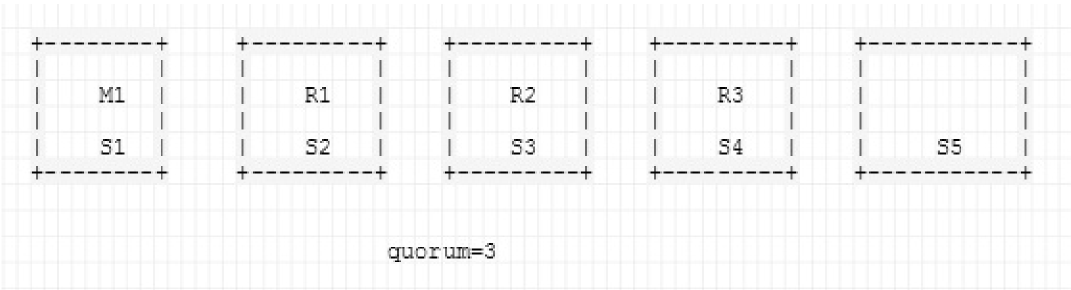
检查集群状态。
```

如果sentinel monitor的quorum设置为2，则宕机一台后再宕机，此时还剩余两台，存在两个sentinel，两个slave。由于quorum为2，必须有>=max(quorum, num(sentinels)/2 +1)=max(2,2) =2个的sentinel都同意其中某一个sentinel主持failover，因此此时存在sentinel可主持切换，因此测试表明，新的master被选出来。

但是设置为2有一个危险就是如果出现如下的网络隔离状况：



集群就会脑裂，就会出现两个master。因此，生产上为了万无一失，宁可牺牲掉一定的高可用容错度也要避免脑裂。如果希望两台宕机依然可以切换，最好的方案不是降低quorum而是增多sentinel的个数，这个建议也是antirez在stackoverflow中回答一个人的提问时给的建议（http://stackoverflow.com/questions/27605843/redis-sentinel-last-node-doesnt-become-master#）。如下场景测试：



此时其中两台宕机，必须有>=max(quorum, num(sentinels)/2 +1)=max(3,3) =3个的sentinel都同意其中某一个sentinel主持failover，因此此时存在sentinel可主持切换，测试结果表明此种部署方案可以正常切换。

11.1.4.12 Master hang死测试

由于我们的sentinel down-after-milliseconds为3100，即3.1s，因此在master上执行： debug sleep 3.0，系统不会切换，但是执行debug sleep 3.7或者更大的数值，系统就会判定为主sdown，进而变为odown随后发起投票切换。很难模拟取消odown的，因为时间差很短。

11.1.4.13 附：sentinel.conf被修改后的含义

port 26379 dir "/var/lib/redis/tmp" sentinel monitor mymaster 192.168.65.128 6379 2 sentinel config-epoch mymaster 18 ###确认mymater SDOWN时长 sentinel leader-epoch mymaster 18 ###同时一时间最多18个slave可同时更新配置,建议数字不要太大,以免影响正常对外提供服务 sentinel known-slave mymaster 192.168.65.129 6379 ###已知的slave sentinel known-slave mymaster 192.168.65.130 6379 ###已知的slave sentinel known-sentinel mymaster 192.168.65.130 26379 be964e6330ee1eaa9a6b5a97417e866448c0ae40 ###已知slave的唯一id sentinel known-sentinel mymaster 192.168.65.129 26379 3e468037d5dda0bbd86adc3e47b29c04f2afe9e6 ###已知slave的唯一id sentinel current-epoch 18 ####当前可同时同步的salve数最大同步阀值

11.1.4.14 附：sentinel事件含义

```
+reset-master <instance details> -- 当master被重置时.
+slave <instance details> -- 当检测到一个slave并添加进slave列表时.
+failover-state-reconf-slaves <instance details> -- Failover状态变为reconf-slaves状态时
+failover-detected <instance details> -- 当failover发生时
+slave-reconf-sent <instance details> -- sentinel发送SLAVEOF命令把它重新配置时
+slave-reconf-inprog <instance details> -- slave被重新配置为另外一个master的slave，但数据复制还未发生时。
+slave-reconf-done <instance details> -- slave被重新配置为另外一个master的slave并且数据复制已经与master同步时。
-dup-sentinel <instance details> -- 删除指定master上的冗余sentinel时 (当一个sentinel重新启动时，可能会发生这个事件).
+sentinel <instance details> -- 当master增加了一个sentinel时。
+sdown <instance details> -- 进入SDOWN状态时;
-sdown <instance details> -- 离开SDOWN状态时。
+odown <instance details> -- 进入ODOWN状态时。
-odown <instance details> -- 离开ODOWN状态时。
+new-epoch <instance details> -- 当前配置版本被更新时。
+try-failover <instance details> -- 达到failover条件，正等待其他sentinel的选举。
+elected-leader <instance details> -- 被选举为去执行failover的时候。
+failover-state-select-slave <instance details> -- 开始要选择一个slave当选新master时。
no-good-slave <instance details> -- 没有合适的slave来担当新master
selected-slave <instance details> -- 找到了一个适合的slave来担当新master
failover-state-send-slaveof-noone <instance details> -- 当把选择为新master的slave的身份进行切换的时候。
failover-end-for-timeout <instance details> -- failover由于超时而失败时。
failover-end <instance details> -- failover成功完成时。
switch-master <master name> <oldip> <oldport> <newip> <newport> -- 当master的地址发生变化时。通常这是客户端最感兴趣的消息了。
+tilt -- 进入Tilt模式。
-tilt -- 退出Tilt模式。
```

11.1.5 其他问题

11.1.5.1 只读性

主从复制架构下，默认Slave是只读的，如果写入则会报错：

```
127.0.0.1:6379> set foo bar
(error) READONLY You can't write against a read only slave.
```

注意这个行为是可以修改的，虽然这样的修改没有意义：

```
127.0.0.1:6379> CONFIG SET slave-read-only no
OK
127.0.0.1:6379> set foo bar
OK
```

11.1.5.1 事件通知

在sentinel中，如果出现warning以上级别的事件发生， 是可以通过如下配置进行脚本调用的（对于该脚本redis启动用户需要有执行权限）：

```
sentinel notification-script mymaster /redis/script/notify.py
```

比如说，我们希望在发生这些事件的时候进行邮件通知，那么，notify.py就是一个触发邮件调用的东东，传入第一个参数为事件类型，第二个参数为事件信息：

```
#!/bin/python

from sendmail import send_mail
import sys

event_type = sys.argv[1]
event_desc = sys.argv[2]
mail_content = event_type + ":" + event_desc

send_mail("xxxx@qq.com",
          ["xxxxx@cmbc.com.cn","xxxx@gmail.com"],
          "Redis Sentinel Event Notification Mail",
          mail_content,
          cc=["xxx@gmail.com","xxx@139.com"],
          bcc=["xxxx@qq.com"]
        )
```

有两个注意事项： 1） 这个时候如果集群发生了切换会产生很多事件，此脚本是在每一个事件发生时调用一次，那么你将短时间收到很多封邮件，加上很多的邮件网关是不允许在一个短时间内发送太多的邮件的，因此这个仅仅是一个示例，并不具备实际上的作用。 2） 一般我们会采用多个sentinel，只需在一个sentinel上配置即可，否则将同一个消息会被多个sentinel多次处理。

附sendmail模块代码：

```
import smtplib
import os
from email.mime.multipart import MIMEMultipart
from email.mime.application import MIMEApplication
from email.mime.base import MIMEBase
from email.mime.text import MIMEText
from email.utils import formatdate
from email import Encoders
from email.message import Message
import datetime

def send_mail(fromPerson,toPerson, subject="", text="",files=[], cc=[], bcc=[]):
    server = "smtp.qq.com"
    assert type(toPerson)==list
    assert type(files)==list
    assert type(cc)==list
    assert type(bcc)==list

    message = MIMEMultipart()
    message['From'] = fromPerson
    message['To'] = ', '.join(toPerson)
    message['Date'] = formatdate(localtime=True)
    message['Subject'] = subject
    message['Cc'] = ','.join(cc)
    message['Bcc'] = ','.join(bcc)
    message.attach(MIMEText(text))

    for f in files:
        part = MIMEApplication(open(f,"rb").read())
        part.add_header('Content-Disposition', 'attachment', filename=filename)
        message.attach(part)

    addresses = []
    for x in toPerson:
        addresses.append(x)
    for x in cc:
        addresses.append(x)
    for x in bcc:
        addresses.append(x)

    smtp = smtplib.SMTP_SSL(server)
    smtp.login("xxxx@qq.com","xxxx")
    smtp.sendmail(message['From'],addresses,message.as_string())
    smtp.close()
```

最佳实践：采用ELK（Elastic+Logstash+Kibana）进行日志收集告警（ElastAlert用起来不错），不启用这个事件通知功能。如果你的环境中没有ELK，或者启动一个Tcp Server进程，notify脚本将事件通过tcp方式吐给这个server，该Server收集一批事件后再做诸如发邮件的处理。

11.1.5.2 虚拟IP切换

在sentinel进行切换时还会自动调用一个脚本（如果设置的话），做一些自动化操作，比如如果我们需要一个虚拟IP永远飘在Master上（这个VIP可不是被应用用来连接redis 的，用过的人都知道连接redis sentinel并不依赖于VIP的），那么可以在sentinel配置文件中配置：

```
sentinel client-reconfig-script mymaster /redis/script/failover.sh
```

在发生主从切换，Master发生变化时，该脚本会被sentinel进行调用，调用的参数如其配置文件所描述的：

```
# The following arguments are passed to the script:
#
# <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>
#
# <state> is currently always "failover"
# <role> is either "leader" or "observer"
#
# The arguments from-ip, from-port, to-ip, to-port are used to communicate
# the old address of the master and the new address of the elected slave
# (now a master).
```

因此，我们可以在failover.sh中进行判断，如果该脚本所运行的主机IP等于新的Master IP，那么将VIP加上，如果不等于，则该机器为Slave，就去掉VIP。通过这种方式进行VIP的切换：

```
#!/bin/sh
_DEBUG="on"
DEBUGFILE=/tmp/sentinel_failover.log
VIP='192.168.2.120'
MASTERIP=${6}
MASK='24'
IFACE='eno33554960'
MYIP=$(ip -4 -o addr show dev ${IFACE}| grep -v secondary| awk '{split($4,a,"/");print
 a[1]}')

DEBUG () {
        if [ "$_DEBUG" = "on" ]; then
                echo `$@` >>  ${DEBUGFILE}
        fi
}

set -e
DEBUG date
DEBUG echo $@
DEBUG echo "Master: ${MASTERIP} My IP: ${MYIP}"
if [ ${MASTERIP} = ${MYIP} ]; then
        if [ $(ip addr show ${IFACE} | grep ${VIP} | wc -l) = 0 ]; then
                /sbin/ip addr add ${VIP}/${MASK} dev ${IFACE}
        DEBUG date
                DEBUG echo "/sbin/ip addr add ${VIP}/${MASK} dev ${IFACE}"
        DEBUG date
        DEBUG echo "IP Arp cleaning: /usr/sbin/arping -q -f -c 1 -A ${VIP} -I ${IFACE}
"
                /usr/sbin/arping -q -f -c 1 -A ${VIP} -I ${IFACE}
        DEBUG date
        DEBUG echo "IP Failover finished!"
        fi
        exit 0
else
        if [ $(ip addr show ${IFACE} | grep ${VIP} | wc -l) != 0 ]; then
                /sbin/ip addr del ${VIP}/${MASK} dev ${IFACE}
                DEBUG echo "/sbin/ip addr del ${VIP}/${MASK} dev ${IFACE}"
        fi
        exit 0
fi
exit 1
```

最早这样的用法是一个日本人写的blog，请参见：http://blog.youyo.info/blog/2014/05/24/redis-cluster/

11.1.5.3 持久化动态修改

其实相对于VIP的切换，动态修改持久化则是比较常见的一个需求，一般在一主多从多Sentinel的HA环境中，为了性能常常在Master上关闭持久化，而在Slave上开启持久化，但是如果发生切换就必须有人工干预才能实现这个功能。可以利用client-reconfig-script自动化该进程，无需人工守护，我们就以RDB的动态控制为例： Sentinel配置文件如下：

```
sentinel client-reconfig-script mymaster /redis/script/rdbctl.sh
```

rdbctl.sh源代码：

```
#!/bin/bash

_DEBUG="on"
DEBUGFILE="/smsred/redis-3.0.4/log/sentinel_failover.log"
MASTERIP=${6}
MASTERPORT=${7}
SLAVEIP=${4}
SLAVEPORT=${5}
MASK='24'
IFACE='bond0'
MYIP=$(ip -4 -o addr show dev ${IFACE}| grep -v secondary| awk '{split($4,a,"/");print
 a[1]}')


DEBUG () {
        if [ "$_DEBUG" = "on" ]; then
                echo `$@` >>  ${DEBUGFILE}
        fi
}




set -e
DEBUG date
DEBUG echo $@
DEBUG echo "===Begin Failover==="
#If Master

if [ ${MASTERIP} = ${MYIP} ]; then
       #Disable RDB
       redis-cli -h ${MYIP} -p ${MASTERPORT} -a c1m2b3c4 config set save ""
       DEBUG echo ${MYIP}
       DEBUG echo "Disable Master RDB:" ${MYIP} ${MASTERPORT}
        DEBUG echo "===End Failover==="
        exit 0

#Or Slave
else
        echo "test5" >> $DEBUGFILE
       redis-cli -h ${MYIP} -p ${SLAVEPORT} -a c1m2b3c4 config set save "900 1 300 10
60 100000000"
       DEBUG echo ${MYIP}
       DEBUG echo "Enable Slave RDB:" ${MYIP} ${SLAVEPORT}
        DEBUG echo "===End Failover==="
        exit 0
fi

exit 1
```

原理和VIP切换一节基本一致，不再赘述。

11.1.5.4 Sentinel最大连接数

1. 问题描述

某准生产系统，测试运行一段时间后程序和命令行工具连接sentinel均报错，报错信息为：

```
jedis.exceptions.JedisDataException: ERR max number of clients reached
```

此时应用创建redis新连接由于sentinel已经无法响应而无法找到master的IP与端口，因此无法连接redis，并且此时如果发生redis宕机亦无法进行生产切换。

2. 问题初步排查过程

首先，通过netstat对sentinel所监听端口26379进行连接数统计，此时连接则报错。如下图：

```
smsred@MSMSRED1:~/redis-3.0.4/conf> netstat -an | grep 26379 | wc -l
10014
smsred@MSMSRED1:~/redis-3.0.4/conf> clis info
ERR max number of clients reached
```

通过sentinel服务器端统计发现，redis sentinel 的连接中大部分都是来自于两台非同网段（中间有防火墙）的应用服务器连接（均为Established状态），并且来自其的连接也大约半个小时后稳步增加一次，而同网段的应用服务器连接sentinel的连接数基本保持一致。排除了应用的特殊性（采用的jedis版本和封装的工具类都是一样的）后，初步判断此问题与网络有关，更详细的说是连接数增加与防火墙切断连接后的重连有关。

3. 问题查证过程

此问题分为两个子问题： 1) 防火墙将TCP连接设置为无效时sentinel服务器为何没有断开连接，保持Established状态？ 2) 为何连接数还会不断增加？

对于问题1)，TCP在三次握手建立连接时OS会启动一个Timer来进行倒计时，经过一个设定的时间（这个时间建立socket的程序可以设置，如果没有设置则采用OS的参数tcp_keepalive_time，这个参数默认为7200s，即2小时）后这个连接还是没有数据传输，它就会以一定间隔（程序可以设定，如果没有设置则采用OS的参数tcp_keepalive_intvl，默认为75s）发出N（程序可以设定，如果没有设置则采用OS的参数tcp_keepalive_probes，默认为9次）次Keep Alive包。TCP连接就是通过上述的过程，在没有流量时是通过发送TCP Keep-

Alive数据包，然后对方回应TCP Keep-Alive ACK来确定信道是否还在真实连接。通过查看Sentinel源代码，其默认是不开启Keepalive的（而jedis默认是开启的），并且默认对于不活动的连接也不会主动关闭的（timeout默认为0）。

对于防火墙，通过翻阅防火墙技术资料（详见下列描述，摘自：《Junos Enterprise Switching: A Practical Guide to Junos Switches and Certification》），我司采用的Juniper防火墙对于没有流量的TCP连接默认是30分钟，30分钟内没有流量就会断掉链路，而不会发送TCP Reset，同时在防火墙策略上并没有开长连接，使用的即为此默认设置。

A Word on Session Timeouts

The flow-based nature of JUNOS software with enhanced services results in a need to age out inactive flows to ensure that flow state does not grow without bounds. The default settings result in TCP-based session timeouts of 1,800 seconds, or 30 minutes. Additionally, the default settings do not reset (clear) TCP sessions upon age out. This can result in the sensation of a "hung terminal" session, where you find the terminal session unresponsive—as though the router had crashed. This behavior differs from that of JUNOS software, which has no such session timeouts given its packet-based forwarding paradigm.

There are a few ways to minimize these issues. First, you can add the tcp-rst option to the management traffic's ingress zone. This results in the TCP session clearing upon timeout, which leaves no doubt as to the connection status, thus minimizing the sensation of a hung terminal session. Another option is to configure longer session timeouts for local host management traffic, which is practical only when you have a specific policy for this management traffic; that is, you are not in router context with an accept-

因此在防火墙每半个小时将连接置为无效时，sentinel同时又禁止了Keepalive（因为默认设置Keepalive为0，即disable发送Keepalive包）。应用服务器的jedis虽然开启了keepalive，但是它发送的keepalive包由于防火墙已经将此链路标记为无效，而无法发送到sentinel端，而且jedis由于采用了OS默认参数，因此需要等待tcp_keepalive_time（2小时）后才启动发送Keep Alive包进行探活的，在tcp_keepalive_time+tcp_keepalive_intvl*tcp_keepalive_probes=7895s=131.58分钟后，jedis端才会认定这个连接断掉而清理掉这个连接。简单的说就是jedis会在很长一段时间后才会发keepalive包，并且这个包也是发不到sentinel上的，而sentinel本身也不会发送keepalive包，所以从sentinel这端看连接一直存在，而从jedis那端看7895s之后就会清理一次连接。这也解释了为什么防火墙将TCP连接断开后，sentinel端的连接并没有释放。

对于问题2)，翻阅jedis源代码，jedis通过连接sentinel并pubsub来监听集群事件，以确定是否发生了切换，并且拿到新的master 地址和端口。如果断开则会5秒后尝试重连（JedisSentinelPool.java）。

```
if (running.get()) {
  log.log(Level.SEVERE, "Lost connection to Sentinel at " + host + ":" + port
      + ". Sleeping 5000ms and retrying.", e);
  try {
    Thread.sleep(subscribeRetryWaitTimeMillis);
  } catch (InterruptedException e1) {
    log.log(Level.SEVERE, "Sleep interrupted: ", e1);
```

因此，这是导致连接数不断上升的原因。 综上，防火墙相对频繁的断开和服务器不断重连导致在一个相对较短的时间内连接骤增，造成到达sentinel最大连接数，sentinel 的最大连接数在redis.h中定义，为10000：

```
#define REDIS_MAX_CLIENTS 10000
```

4. 问题解决过程

此系统由于访问关系与网段规划间的安全问题，必须跨越防火墙，因此试图从配置角度解决此问题。

首先，联系网络相关同事，进行网络变更，开启从应用服务器到sentinel的链路相对的长连接，即无流量超时而断开的时间设置为8小时。以此手段降低断开频率，以便缓解短时间内不断重试连接造成的sentinel连接增长。

然后，通过阅读redis源代码（net.c），发现，sentinel也采用了redis 所有参数设置（通过config.c的函数void loadServerConfigFromString(char *config)）。因此，通过设置redis 的下列两个参数可以解决这个问题，第一个参数是TCP Keepalive参数，此参数默认为0，也就是不发送keepalive。也就是改变OS默认的tcp_keepalive_time参数（在Unix C的socket编程中TCP_KEEPIDLE参数对应OS 的tcp_keepalive_time参数）。

```
val = interval;
if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPIDLE, &val, sizeof(val)) < 0) {
    __redisSetError(c,REDIS_ERR_OTHER,strerror(errno));
    return REDIS_ERR;
}
```

该参数的官方解释为：

```
# TCP keepalive.
#
# If non-zero, use SO_KEEPALIVE to send TCP ACKs to clients in absence
# of communication. This is useful for two reasons:
#
# 1) Detect dead peers.
# 2) Take the connection alive from the point of view of network
#    equipment in the middle.
#
# On Linux, the specified value (in seconds) is the period used to send ACKs.
# Note that to close the connection the double of the time is needed.
# On other kernels the period depends on the kernel configuration.
#
# A reasonable value for this option is 60 seconds.
```

我们设置为tcp-keepalive 60，加快回收连接速度，从网络断开到连接清理时间缩短为60+75*9=12.25分钟。

同时，通过设置maxclients为65536，增大sentinel最大连接数，使得在上述12.25分钟即使有某种异常导致sentinel连接数增加也不至于到达最大限制。此参数的官方解释为：

```
################################### LIMITS ####################################

# Set the max number of connected clients at the same time. By default
# this limit is set to 10000 clients, however if the Redis server is not
# able to configure the process file limit to allow for the specified limit
# the max number of allowed clients is set to the current file limit
# minus 32 (as Redis reserves a few file descriptors for internal uses).
#
# Once the limit is reached Redis will close all the new connections sending
# an error 'max number of clients reached'.
#
maxclients 10000
```

对于redis 的timeout参数，由于启用这个参数有程序微小开销（会调用redis.c中的int clientsCronHandleTimeout(redisClient *c, mstime_t now_ms)），决定保持默认为0，而通过上述参数使用OS进行连接断开。

5. 问题解决结果

通过开发、网络和数据库团队的协同努力，配置上述参数和修改防火墙策略后，手动增加sentinel进程，超过原默认最大连接数10000后sentinel可以正常访问操作，并且通过tcpdump进行抓包，在指定时间内（1分钟），就有KeepAlive包对每个sentinel TCP连接进行探活，经过观察sentinel连接稳定，再未出现短时间内暴涨的情况。

6. 问题后续

在redis中默认不开启keepalive就是为了尽可能减小网络负载，榨干网络性能，尽可能达到redis的。在后续的程序运行中，如果发现网络是瓶颈时（在相当长的一段时间内不会），可以加大sentinel的keepalive参数，减小keepalive数据包的传输，这个修改是不影响redis对外服务的。

参考文档： http://www.tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO/

附录：如何用TCPDUMP进行keep alive抓包

```
tcpdump -pni bond0 -v "src port 26379 and ( tcp[tcpflags] & tcp-ack != 0 and ( (ip[2:2
] - ((ip[0]&0xf)<<2) ) - ((tcp[12]&0xf0)>>2) ) == 0 ) "
```

7. 问题再后续

我们后来在这个应用上发现一旦网络有抖动，sentinel的连接增加就回大幅度增加，后来通过jmap查看sentinelpool的实例竟然多达200多个，也就是说这个就是程序的问题，在sentinelpool上不应该多次实例化，而是采用已有连接进行重连。